

**Politechnika Wrocławskas
Wydział Informatyki i Telekomunikacji**

Kierunek: **Informatyka Algorytmiczna (INA)**

Specjalność: -

**PRACA DYPLOMOWA
INŻYNIERSKA**

**Implementacja programu GRAPHPLAN do
planowania akcji z wykorzystaniem
programowania ograniczeń**

Radosław Wojtczak

Opiekun pracy
dr Przemysław Kobyłański

Słowa kluczowe:
Planowanie
Grafy
Programowanie ograniczeń
Sztuczna inteligencja

Streszczenie

Obiektem badań poniższej pracy jest metodologia planowania o nazwie "**GRAPHPLAN**", bazująca na formalnym języku **STRIPS**, skonstruowanym z myślą o przedstawianiu rozpatrywanych światów w formie stanów (początkowego jak i końcowego), oraz akcji między nimi, które wpływają na jego zmiany. Esencją GRAPHPLAN'u jest wykorzystanie struktury zwanej **grafem planującym** w trakcie ustalania optymalnego planu transformacji stanu początkowego w stan końcowy w ustalonej przestrzeni przy wykorzystaniu wcześniej zdefiniowanych operatorów.

Praca ta składa się z formalnego opisu przytoczonego algorytmu, przedstawienia przykładów zastosowania, implementacji, której wynikiem jest graf przedstawiający optymalny plan wykonywanych operacji oraz zmian, jakie dzięki nim zachodzą w przedstawionym świecie, omówienie opcjonalnych rozszerzeń, które w zależności od sytuacji mogą wpływać na efektywność algorytmu oraz przeprowadzonych testów, których zadaniem jest wskazanie mocnych, jak i słabych stron przedmiotu badań.

Dodatkowym aspektem pracy jest przedstawienie koncepcji o nazwie **programowanie ograniczeń**, która została wykorzystana w celu usprawnienia osiągnięć czasowych algorytmu. Wprowadzono formalne definicje programowania ograniczeń wraz z jego własnościami oraz przedstawiono w jaki sposób rzeczone podejście do programowania zostało zawarte w implementacji badanego algorytmu.

Abstract

The main goal of given thesis is to implement an algorithm for automated planning called "**GRAPHPLAN**", which is based on formal language named **STRIPS**, that uses states (with focus on two special states: initial and goal states) and actions, which are used to make changes in the described world. The core of GRAPHPLAN is the usage of a structure called a **planning graph** which greatly reduces the amount of search needed during creation of a plan from initial states to goal states.

Presented thesis consists of detailed description of algorithm, application examples, implementation, which uses a graph to show the optimal plan constructed from defined actions and changes made after each action in a given world, provides optional extensions, which depending on circumstances can impact on an efficiency of an algorithm and tests, which main goal is to show benefits and potential drawbacks of usage given methodology of planning.

Additionally, an integral part of this thesis is a concept called **constraint programming**, which heavily impacts on performance of described algorithm. This paper provides an exhaustive description with formal definitions and examples of a given way of programming and shows how constraint programming is integrated into planning graph.

Spis treści

Spis rysunków	IV
Spis tabel	V
Spis kodów	VII
Wstęp	1
1 Wprowadzenie	3
1.1 Planowanie	3
1.2 Planowanie przy użyciu komputerów	5
1.2.1 STRIPS	5
1.2.2 Rozwiązywanie ludzkich problemów	6
1.2.3 Liniowy i częściowy porządek	6
1.2.4 ADL i PDDL	7
1.2.5 Nowoczesne rozwiązania	7
1.2.6 Miejsce GRAPHPLAN'u	7
2 GRAPHPLAN	9
2.1 Wprowadzenie	9
2.2 Warunki początkowe	9
2.3 Akcje	10
2.3.1 Typy akcji	11
2.4 Definiowanie świata	12
2.5 Warstwy grafu	13
2.6 Równoległość	14
2.6.1 Wzajemne wykluczanie	15
2.7 Wyszukiwanie planu	18
2.8 Własności GRAPHPLAN'u	22
2.8.1 Złożoność obliczeniowa	23
2.8.2 Problem stopu	23
3 Programowanie ograniczeń	25
3.1 Wprowadzenie	25
3.2 Pojęcie ustalenia i spójności	26
3.3 Ograniczenia globalne	27
3.4 Wyszukiwanie rozwiązań	27
3.5 Programowanie w logice z ograniczeniami	28
3.6 Obrazowe przykłady	29
3.6.1 Problem plecakowy	29
3.6.2 Krypto-arytmetyczna łamigłówka	30
3.7 Wykorzystanie w algorytmie	31
4 Implementacja	33

4.1	Opis komponentów i ich połączeń	33
4.2	Implementacja algorytmu	34
4.3	Generowanie grafów	40
4.3.1	Struktura pliku tekstowego	41
4.3.2	Graf pełny	41
4.3.3	Graf prosty	46
4.4	Interfejs użytkownika	48
4.5	Uruchomienie algorytmu z linii komend	50
5	Instalacja i wdrożenie	53
5.1	Instalacja pakietu SWI-Prolog	53
5.2	Instalacja języka Python	53
5.3	Dokumentacja	54
6	Testy	55
6.1	15	55
6.1.1	Wprowadzenie	55
6.1.2	Teoria	56
6.1.3	Przykład	57
6.1.4	Szczegóły implementacyjne	59
6.1.5	Wyniki	61
6.1.6	Młodsza siostra- ósemka	63
6.1.7	Wyniki dla 8	63
6.1.8	Wnioski	64
6.2	CargoBot	64
6.2.1	Wprowadzenie	64
6.2.2	Szczegóły implementacyjne	64
6.2.3	Przykład	66
6.2.4	Wyniki	68
6.2.5	Wnioski	68
6.3	Przemieszczanie w przestrzeni	68
6.3.1	Wprowadzenie	68
6.3.2	Przykład	68
6.3.3	Szczegóły implementacyjne	68
6.3.4	Wyniki	68
6.3.5	Wnioski	68
6.4	Wieża Hanoi	68
6.4.1	Wprowadzenie	68
6.4.2	Przykład	68
6.4.3	Szczegóły implementacyjne	68
6.4.4	Wyniki	68
6.4.5	Wnioski	68
6.5	Osiem Hetmanów	68
6.5.1	Wprowadzenie	68
6.5.2	Przykład	68
6.5.3	Szczegóły implementacyjne	68
6.5.4	Wyniki	68
6.5.5	Wnioski	68
Podsumowanie	69	
Bibliografia	71	
A Zawartość płyty CD	73	

Spis rysunków

1.1	Przeniesienie klocka z jednej półki na drugą jako przykład sytuacji dla której istnieje możliwość utworzenia planu. Po lewej stronie od strzałki znajduje się stan początkowy, natomiast po prawej- oczekiwany cel. Naturalną akcją w przedstawionym świecie jest akcja <i>przenieś</i> , która zadanego klocka przenosi z jednej półki na drugą.	4
1.2	Problem komiwojażera (Travelling Salesman Problem, TSP). Na rysunku pomocniczym liczbę symbolizującą wagę krawędzi. Węzeł oznaczony kolorem czerwonym odpowiada węzłowi startowemu, do którego należy wrócić. Długości krawędzi nie zachowują wskazanych przez wagę proporcji.	5
2.1	Najogólniejsza forma grafu planującego. Składa się on z węzłów, zwanych stanami oraz krawędzi zwanych akcjami. Docelowo poszczególne stany oraz akcje są parami różne, jednak mogą zajść sytuacje, gdy powtórzenie któregoś z komponentów będzie wymagane do uzyskania odpowiedniego celu.	9
2.2	Przykładowy moment startowy przyszłego planu. Za pomocą okręgów oznaczono roboty, natomiast poprzez kwadraty oznaczone są kafelki- miejsca, po których mogą poruszać się roboty.	10
2.3	Obrazowe przedstawienie ruchu robota B z kafelka 3 na kafelek 2	11
2.4	Obrazowe przedstawienie wszystkich akcji w pierwszym kroku algorytmu, akcje opisane przy użyciu czcionki o kolorze zielonym symbolizują akcje podtrzymujące, natomiast o kolorze czerwonym- akcje aktywne	12
2.5	Graficzne przedstawienie relacji sąsiedztwa dla kafelka numer 4. Kolorem kremowym oznaczono miejsca, z którymi sąsiaduje, natomiast pomarańczowym- te, z którymi nie sąsiaduje. Oznacza to, iż do kafelków pomarańczowych nie można dostać się wykonując jeden ruch	13
2.6	Modyfikacja do grafu planującego wprowadzona w poprzednich podrozdziałach. Wprowadzono zależności akcji od stanu poprzedniego, oraz stanu następnego od akcji.	13
2.7	Przedstawienie sposobu wyznaczania warstw w grafie planującym. Poziom stanów oraz akcji wchodzący w skład danej warstwy wyróżniony jest poprzez konkatenację nazwy oraz liczby symbolizującej numer warstwy.	14
2.8	Przykład możliwości zastosowania równoległości w planowaniu działania. Należy zauważać, iż ruchy robotów w żaden sposób ze sobą nie kolidują.	15
2.9	Przykład świata, w którym wprowadzenie równoległości dla pierwszej warstwy algorytmu jest niemożliwe. Dwa roboty próbują przejść na ten sam kafelek w tej samej jednostce czasu, co z perspektywy kafelka powoduje konflikt. Odpowiednio kolorami: czerwonym i niebieskim oznaczono roboty oraz kafelki, które są ich celem.	16
2.10	Urywek planu przedstawiający sytuację, w której dwa stany powstają w sposób niespójny. Zgodnie z założeniami akcje oznaczone są przy pomocy prostokątów, stany- okręgów, miedzy akcjami a stanami czerwone linie symbolizują które stany są efektami których akcji, natomiast linie niebieskie między akcjami symbolizują powstałe między nimi <i>mutex</i>	17
2.11	Przykład wykluczania się akcji aktywnych z podtrzymującymi	17
2.12	Przykład świata, w którym każdy z robotów próbuje przesunąć się na sąsiadujący z lewej strony klocki. Uwidoczniona sytuacja jest przykładem wykluczających się akcji	18
2.13	Sytuacja początkowa świata, dla którego odbędzie się przykładowe generowanie planu. Kolorem niebieskim zaznaczono kafelek docelowy robota	19
2.14	Warunki początkowe omawianego świata. Obrazki ułatwiające analizę przykładu w całości zostały wykonane przez oprogramowanie utworzone na rzecz pracy.	19

2.15 Wygenerowane akcje na podstawie warunków początkowych	20
2.16 Wygenerowany drugi stan grafu bezpośrednio wynikający z pierwszego poziomu akcji	20
2.17 Przykład kolejnej iteracji algorytmu	21
2.18 Graf planujący z zaznaczonymi akcjami aktywnymi prowadzącymi do uzyskania wskazanego celu	21
2.19 Przykład świata, w którym osiągnięcie celu, którym jest kafelek oznaczony kolorem takim jak kolor użytej czcionki do oznaczenia sygnatury robota, jest niemożliwe ze względu na absencję jednego z kafelków, będących łącznikiem, między robotem a jego celem.	23
3.1 Przykładowa łamigłówka krypto-arytmetyczna	30
3.2 Przykładowe rozwiązywanie wskazanej łamigłówki	31
4.1 Zależności między komponentami	33
4.2 Główna pętla algorytmu	34
4.3 Urywek przykładowego pliku tekstowego wygenerowanego przez algorytm	41
4.4 Diagram przepływu dla generowania pojedynczego grafu pełnego	42
4.5 Szczegółowy diagram przepływu dla parsowania pliku tekstowego	43
4.6 Przykład grafu pełnego dla prostego przypadku	45
4.7 Przykład grafu pełnego dla złożonego przypadku	46
4.8 Diagram przepływu dla generowania pojedynczego grafu prostego	46
4.9 Przykład grafu prostego dla prostego przypadku	47
4.10 Przykład grafu pełnego dla złożonego przypadku	48
4.11 Okno startowe programu	48
4.12 Widok po wybraniu opcji "3x3"	49
4.13 Widok po wybraniu opcji CargoBOT	49
4.14 Proces definiowania świata	50
4.15 Stan po wygenerowaniu planu	50
4.16 Zrzut ekranu prezentujący przykładowe uruchomienie algorytmu z linii komend	51
6.1 Wygląd układanki piętnastki wygenerowany przy pomocy zaimplementowanej w ramach pracy warstwy graficznej	55
6.2 Losowa rozwiązywalna permutacja układanki	56
6.3 Piętnastka w formie obrazkowej. Źródło: http://mypuzzlecollection.blogspot.com/2012/08/mc-escher-birds-fish-and-turtles.html	56
6.4 Przykładowe startowe ułożenie przesuwanki	57
6.5 Akcje rozpatrywane przez algorytm w danym kroku	58
6.6 Uproszczony graf planujący wygenerowany przez algorytm GRAPHPLAN przedstawiający stan każdego kafelka w danej warstwie. Węzły wypełnione kolorem szarym obrazują stany, które są warunkami zajścia jak i efektami wykonywanej w danej warstwie akcji	58
6.7 Obrazowe rozwiązywanie na podstawie wygenerowanego grafu	59
6.8 Przyporządkowywanie klockom odpowiednich liter	59
6.9 Przykładowa łamigłówka, źródło zdjęcia: https://i4ds.github.io/CargoBot/?state=3 [ostatni dostęp:06.12.2022]	66
6.10 Reprezentacja łamigłówki w aplikacji	66
6.11 Rozwiązywanie w formie listy kroków zaprezentowane w aplikacji	67
6.12 Reprezentacja łamigłówki w aplikacji	67

Spis tabel

3.1	Otrzymane wyniki dla znalezienia pierwszego rozwiązania łamigłówki	31
4.1	Tabela sygnatur linii w pliku tekstowym	41
6.1	Przykładowe czasy dla wybranych przykładów	62
6.2	Przykładowa liczba wnioskowań dla wybranych przykładów	62
6.3	Średnia liczba wnioskowania oraz czasów wykonania	62
6.4	Przykładowe czasy dla wybranych przykładów	63
6.5	Przykładowa liczba wnioskowań dla wybranych przykładów	63
6.6	Średnia liczba wnioskowania oraz czasów wykonania	64



List of Listings

1	Metoda wypisująca liczby gdy ich suma jest większa od 0	28
2	Implementacja rozwiązania łamigłówki krypto-arytmetycznej bez użycia programowania ograniczeń	30
3	Implementacja rozwiązania łamigłówki krypto-arytmetycznej z użyciem programowania ograniczeń	31
4	Implementacja predykatu call_plan/2	35
5	Implementacja predykatu create_plan/2	35
6	Implementacja predykatu graphplan/4	36
7	Kod źródłowy implementacji predykatu satisfied/2	36
8	Implementacja predykatu extract_plan/2	36
9	Implementacja predykatu expand/4	37
10	Implementacja predykatu add_actions/6	37
11	Implementacja predykatu add_effects/4	38
12	Implementacja predykatu mutex_state/1	38
13	Implementacja predykatu mutex_single/2	39
14	Implementacja predykatu mutex/2	39
15	Implementacja predykatu mutex_for_action/3	39
16	Implementacja predykatu mutex_all_states/3	40
17	Implementacja predykatu apply_mutex/4	40
18	Implementacja parsowania pliku tekstowego	44
19	Implementacja funkcji generateLayer()	44
20	Instalacja pakietu SWI-PROLOG z poziomu linii komend	53
21	Komenda sprawdzająca wersję zainstalowanego języka python	53
22	Instalacja odpowiednich bibliotek dla języka python	54
23	Uruchomienie interfejsu użytkownika	54
24	Implementacja predykatu preconditions/2 dla przesuwanki	60
25	Implementacja predykatu effects/2 dla przesuwanki	60
26	Implementacja predykatu adjacent/2	60
27	Modelowanie relacji sąsiedztwa	61
28	Implementacja predykatu inconsistent/2	61
29	Implementacja predykatu preconditions/2 dla CargoBOT'a	65
30	Implementacja predykatu effects/2 CargoBOT'a	65
31	Implementacja predykatu object/1	65



Wstęp

Celem pracy jest zaimplementowanie algorytmu do planowania akcji o nazwie **GRAPHPLAN**, który po raz pierwszy został sformalizowany i opisany w pracy pod tytułem **”Fast Planning Through Planning Graph Analysis”**[3] przez Panów: Avrima L. Blum'a i Merricka L. Furst'a.
Praca składa się z sześciu rozdziałów.

W rozdziale pierwszym poruszono aspekty historyczne odnośnie planowania akcji przy użyciu komputerów oraz jaką rolę pełni w niej GRAPHPLAN, dokonano przedstawienia różnicy między liniowym, a częściowym typem planowania oraz przedstawiono nowoczesne rozwiązania stosowane we wskazanej dziedzinie programowania.

W rozdziale drugim poddano dogłębnej analzie implementowany algorytm- dokładnie opisano jego strukturę, warstwy, z których się składa oraz własności, które wyróżniają go na tle innych rozwiązań problemów związanych z planowaniem. W celu łatwiejszego przyswojenia mechanizmów stojących za GRAPHPLAN'em w trakcie opisu wprowadzono liczne proste przykłady wraz z grafikami wygenerowanymi przy pomocy narzędzi stworzonych na potrzeby ów pracy.

W rozdziale trzecim rozwinięto pojęcie programowania ograniczeń, wprowadzając formalną definicję, podstawowe słownictwo niezbędne do zrozumienia idei stojącej za tym sposobem programowania, omówiono benefity płynące z wykorzystania tego podejścia oraz przedstawiono obrazowo schemat funkcjonowania na podstawie prostych przykładów.

Rodzaj czwarty skupia się na szczegółach implementacyjnych: wybranych językach programowania oraz technologiach wykorzystywanych również w warstwach graficznych programu. Dokonano szczegółowego opisu interfejsu użytkownika oraz jego możliwości, połączeń między komponentami oraz ważniejszych funkcji stanowiących trzon pracy.

Rodzaj piąty przedstawia sposób instalacji oraz instrukcję obsługi programu, jak i instalowania wszystkich niezbędnych komponentów wykorzystywanych w pracy, w których skład wchodzą interpreterы jak i kompilatory używanych języków programowania oraz wszystkie biblioteki i moduły.

Rodzaj szósty przedstawia przeprowadzone testy, które badają możliwości algorytmu w wcześniej spersonowanych środowiskach. W tej części została przeprowadzona analiza wydajnościowa algorytmu, weryfikacja wygenerowanych planów pod względem poprawności oraz porównanie otrzymanych wyników z innymi powszechnie wykorzystywanymi metodami planowania. Każdy z testów zawiera w sobie wniosek, w którym odbywa się zbiorcza ocena wszystkich wyżej wymienionych aspektów.

Końcowy rozdział stanowi zbiorcze podsumowanie pracy z komentarzem odnośnie potencjalnych rejonów, w których algorytm mógłby znaleźć swoje zastosowanie.



Rozdział 1

Wprowadzenie

1.1 Planowanie

Każdy człowiek codziennie po wielokroć dokonuje proces określony mianem planowania, często nie zważając na to, jak skomplikowany proces wykonuje. Skonstruowane przez ludzi plany mogą odnosić się do tak trywialnych zagadnień jak utworzenie listy zakupów, która wprost jest generowana przez braki w domowej lodówce oraz upodobania gastronomiczne kupującego, do bardziej abstrakcyjnych form jak *plan na życie*, czy *plan na wygranie meczu*. Przyglądając się planom oraz ich własnościom można wyodrębnić następujące trzy aspekty:

Definicja 1.1 *Warunki początkowe – stan świata przed zastosowaniem akcji. W dalszej części pracy również określane jako stany początkowe.*

- Każdy plan musi mieć jasno zadeklarowane warunki początkowe. Dzięki dokładnej wiedzy o świecie możliwym jest poprawne określenie akcji, przy pomocy których wprowadzane są modyfikacje obecnego stanu aż do otrzymania zadowalających rezultatów. Dla przykładu, firma musi wiedzieć ile oraz jakie palety przybędą na magazyn zanim rozpoczęcie planowanie rozkładu dostawy na magazynie.

Definicja 1.2 *Akcja – działanie zmieniające przedstawiony świat w ścisłe określony sposób. W dalszej części pracy również określana jako operator.*

- Akcje pozwalają na modyfikację przedstawionego świata. Każda z akcji składa się z podmiotu, na który działa oraz czynności, która jest względem wskazanego podmiotu wykonywana. Przykładem dobrze określonej akcji może być przeniesienie klocka z jednego stolika na drugi- składa się ona z podmiotu w postaci klocka, oraz czynności w postaci przenoszenia, które możemy traktować w ogólniejszy sposób jako ruch. Czynności mogą różnić się od siebie w kwestii skomplikowania, najważniejszym jest, aby były określone poprawnie i aby były wykonalne w zdefiniowanym świecie.

Definicja 1.3 *Cel – Oczekiwany stan świata.*

- Kwintesencją każdego planu jest cel, który należy uzyskać. Zwyczajowo plany składają się z celów możliwych do osiągnięcia ze stanu początkowego przy pomocy zdefiniowany operacji, jednakże trzeba wziąć pod uwagę sytuację, w której niemożliwym jest uzyskanie wskazanego celu, szczególnie próbując automatyzować pojęcie planowania.

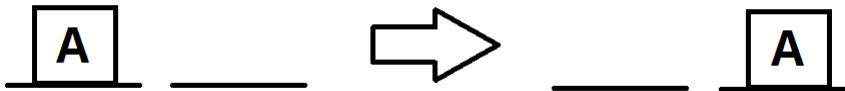
Przy pomocy powyższych definicji możliwym jest sformalizowanie pojęcia stojącego za słowem **plan**.

Definicja 1.4 *Plan – lista akcji, której zastosowanie do stanu początkowego powoduje jego zmianę do stanu określonego w ramach celu.*

Oczywiście nie każdy plan musi być wykonalny. Zdarzają się również sytuacje, w których nie da się utworzyć planu dla zadanych warunków początkowych oraz celu. Z takimi przypadkami algorytm jest sobie w stanie sprawnie poradzić (formalny dowód w dalszej części pracy), jednakże istnieje inna grupa przypadków, która z perspektywy ów metodologii jest niemożliwa do realizacji. Mowa tu o sytuacjach, gdy któraś z akcji zmienia się w akcję **warunkową**.

Definicja 1.5 *Akcja warunkowa to typ akcji, która w zależności od bieżącego stanu świata produkuje inne wyniki*

Przykładem akcji warunkowej jest kopnięcie dmuchanej piłki. W zależności od wiejącego wiatru, które często zmienia się w sposób dynamiczny, przyłożenie tej samej siły do kopnięcia skutkuje różnym punktem końcowym trasy piłki. Ze względu na ograniczenia języka STRIPS plany w takim kontekście są niemożliwe do utworzenia, dlatego w dalszej części pracy wszystkie przedstawione sytuacje będą możliwe do całkowitego opisu przy pomocy nomenklatury STRIPS.



Rysunek 1.1: Przeniesienie klocka z jednej półki na drugą jako przykład sytuacji dla której istnieje możliwość utworzenia planu. Po lewej stronie od strzałki znajduje się stan początkowy, natomiast po prawej- oczekiwany cel. Naturalną akcją w przedstawionym świecie jest akcja *przenieś*, która zadany klocek przenosi z jednej półki na drugą.

Łatwo zauważać na podstawie przykładu 1.1, iż można utworzyć wiele planów, które dla zadanego stanu początkowego osiągają wskazany cel. Dla powyższej sytuacji naturalnym planem jest przeniesienie klocka A z platformy po lewej na platformę po prawej, lecz nie jest to jedyna możliwość. Również satysfakcjonującym planem zgodnie z wprowadzoną wyżej definicją byłaby następująca sekwencja akcji:

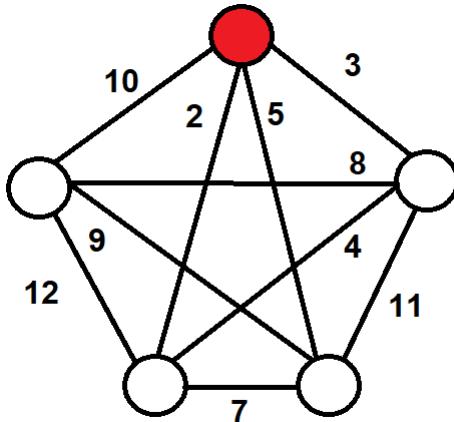
1. Przenieś klocek A z lewej platformy na prawą
2. Przenieś klocek A z prawej platformy na lewą
3. Przenieś klocek A z lewej platformy na prawą

Generowanie rekrecyjne nieskończonych planów poprzez bezużyteczne przestawienia "w miejscu", mimo tego, iż zawiera się w definicji 1.4, nie jest oczekiwany efektem. Proces planowania odbywa się po to, by wykonać transformację świata przy jak najmniejszym nakładzie sił w jak najkrótszym czasie. Z tego względu wprowadzono pojęcie **planu optymalnego**.

Definicja 1.6 *Plan optymalny – Plan o minimalnej liczbie kroków satysfakcjonujący wskazany cel.*

W dalszej części pracy słowo **plan** najczęściej będzie utożsamiane z planem optymalnym.

Wprowadzenie powyższej definicji wiąże się z powstaniem bardzo ważnego pytania: *Jak utworzyć plan optymalny?*. Bazując na przykładach takich jak 1.1 zasadniczo może być myśl, iż generowanie optymalnych planów jest rzeczą prostą. Niech kolejny przykład będzie tego dowodem.



Rysunek 1.2: Problem komiwojażera (Travelling Salesman Problem, TSP). Na rysunku pomocniczym liczby symbolizują wagi krawędzi. Węzeł oznaczony kolorem czerwonym odpowiada węzlowi startowemu, do którego należy wrócić. Długości krawędzi nie zachowują wskazanych przez wagę proporcji.

Problem komiwojażera jest popularnym zagadnieniem optymalizacyjnym, którego istotą jest wskazanie ścieżki o najmniejszej sumie wag krawędzi, która przechodzi dokładnie jeden raz przez każdy wierzchołek i wraca do wierzchołka startowego. Często problem wędrującego komiwojażera przedstawiany jest przy pomocy kuriera oraz domów, które musi odwiedzić (za wierzchołek startowy uważa się magazyn, w którym kurier rozpoczyna swoją pracę). Mimo faktu, iż w przykładzie 1.2 występuje jedynie 5 miejsc, w których musi zatrzymać się kurier, utworzenie optymalnego planu dla przedstawionej sytuacji jest nielada wyzwaniem. Z tego powodu ludzie postanowili skorzystać z potężnych mocy obliczeniowych komputerów przy generowaniu bardziej skomplikowanych planów.

1.2 Planowanie przy użyciu komputerów

1.2.1 STRIPS

W 1971 roku Panowie: Richard Fikes oraz Nils Nilsson z Standford Research Institute (SRI International, jeden z najsłynniejszych na świecie ośrodków badawczych) zdecydowali się na przedstawienie świata nowego podejścia w dziedzinie planowania o nazwie **STRIPS** (STanford Research Institute Problem Solver)^[8]. **STRIPS** rozwiązuje wskazany problem poprzez przeszukiwanie wszystkich stanów świata, aż do momentu gdy znajdzie taki, w którym wskazane cele są spełnione. Ważnym założeniem programu jest istnienie ciągu akcji, który gwarantuje otrzymanie celu. Zadanie to jest realizowane poprzez znalezienie sekwencji operatorów, która konwertuje wymodelowany stan początkowy, w model, w którym wszystkie zdefiniowane cele są spełnione. Definicje operatorów, stanu początkowego oraz celu są niemalże identyczne jak 1.1, 1.2 i 1.3, z tym, że definicja Akcji została w naturalny sposób rozwinięta o ciąg przyczynowo-skutkowy. Zauważono, iż w skład każdej akcji poza samą czynnością wchodzą dwie składowe, nazywane środowiskowymi- warunki zajścia oraz efekty zajścia.

Definicja 1.7 *Warunkiem zajścia akcji jest istnienie odpowiedniej konfiguracji świata, dzięki której akcja może zostać wykonana.*

Definicja 1.8 *Efektem zajścia akcji są zmiany, które zaszły w przedstawionym świecie ze względu na jej wykonanie.*

Mówiąc kolokwialnie, każda akcja ma swoją przyczynę oraz swój skutek. **Przyczyną** akcji w przykładzie 1.1 jest znajdowanie się klocka na lewej platformie. Gdyby klocek A znajdował się na prawej platformie, wykonanie akcji przesunięcia klocka z platformy lewej na prawą nie mogłoby zostać wykonane, natomiastem **efektem** akcji jest przeniesienie klocka na prawą platformę. Łatwo zauważać, iż brak innych obiektów na platformie prawej jest niezbędnym, aby klocek mógł zostać tam przeniesiony. Kolejną naturalną obserwacją jest stwierdzenie, iż przeprowadzenie akcji dodaje nam nowe informacje o świecie w dwóch kontekstach:



- Dodającym – pojawienie się lub podrzymanie danej składowej świata
- Usuwającym – pozbawienie świata danej składowej

Po wykonaniu czynności z przykładu 1.1 wyróżniamy trzy typy nowych informacji:

- Warunek- Obecność klocka na lewej platformie, prawa platforma jest pusta
- Efekt dodający – Obecność klocka na prawej platformie
- Efekt usuwający – Brak obecności klocka na lewej platformie (platforma lewa jest pusta)

W rozważanym podejściu każda z czynności zdefiniowana jest przy pomocy wyżej wskazanych trzech składowych.

Takie zdefiniowane świata okazuje się wystarczające do rozwiązywania problemów pokroju rearanżacji obiektów czy nawigowania w ścisłe zdefiniowanej przestrzeni, czego najlepszym przykładem, jest pierwszy robot do realizacji ów zadań- **Shakey**. Shakey był pierwszym robotem, który dzięki zainstalowanemu oprogramowaniu, posiadał umiejętności analizy własnego otoczenia. Dzięki zaimplementowanemu podejściu STRIPS (oczywiście z odpowiednimi dostosowaniami do sytuacji) był w stanie rozwiązywać problemy z zakresu wyznaczania drogi, czy planowania rozmieszczenia obiektów w pokojach. Ze względu na swoją innowacyjność i przełomowość często jest nazywany archetypem dzisiejszych autonomicznych samochód czy militarnych dronów.

Dzięki swojej roli w rozwoju planowania z użyciem komputerów, STRIPS został dodatkowo wyróżniony od jego nazwy pochodzi język opisu świata korzystający z trójki: stan początkowy, akcja oraz cel. Przez następne lata rozwiązywania w obszarze planowania silnie bazowały na wprowadzonym w powyższej pracy opisie świata.

1.2.2 Rozwiązywanie ludzkich problemów

W 1972 roku Panowie Allen Newell i Herbet Simon opublikowali książkę pod nazwą **Human Problem Solving**, której tłumaczenie znajduje się w tytule opisywanej sekcji. Założenie, odnośnie otrzymywanych przez algorytm rozwiążający problem danych, pozostało zgodne z opisem w sekcji 1.2.1, jednakże autorzy pracy zauważali, iż utworzony w ten sposób obszar działań (*problem space*) może osiągać niebotyczne rozmiary. Chcąc odpowiedzieć na pytanie, w jaki sposób ludzie wybierają odpowiednie operacje w danej sytuacji, zauważali, iż ludzie często wykorzystują *skróty*, które wynikają bezpośrednio z intuicji. Człowiek ze względu na swoją naturę będzie dążył do zużycia jak najmniejszych pokładów energii w celu osiągnięcia wyznaczonego celu. Na tej podstawie rozpoczęło się formowanie technik rozwiązywania problemów zwanych **heurystykami**. Początkowe heurystiki ograniczały się do zastosowania pewnych ograniczeń jeśli chodzi o wyznaczanie zbioru operatorów, na przykład poprzez unikanie powtórzeń (*repeat-state avoidance*) oraz unikanie powrotów do stanów już znanych (*backup avoidance*). Na przestrzeni kolejnych lat pojęcie heurystyki rozwinię się diametralnie, jednakże zachowa swoją pierwotną definicję szukania rozwiązań na skróty lub kolokwialnie mówiąc *na ludzkie oko*.

1.2.3 Liniowy i częściowy porządek

Na początku lat 70 ubiegłego wieku do tworzenia prostych planów wykorzystywano również pojęcie liniowego porządku (*total order*) sekwencji akcji. W tym podejściu, ochrzczonym mianem **planowanie liniowe** (*linear planning*) dla każdego z celów próbowano utworzyć odpowiedni podplan, który go osiągał. Następnie ów plan łączono przy pomocy odpowiedniego porządku akcji w jeden spójny plan. Ze względu na wady tego podejścia, jak długi czas formowania planu oraz możliwość wystąpienia konfliktów między odpowiednimi celami planu, co na początku nie było takie oczywiste, zamiast liniowego porządku zaczęto wykorzystywać porządek częściowy (*partial order*), którego przykładem jest omawiany **GRAPHPLAN**.



1.2.4 ADL i PDDL

Action Description Language (w skrócie **ADL**) zostało zaproponowane przez Edwina Pednault'a w 1987 roku jako usprawnienie języku opisu STRIPS. Ów system automatycznego planowania został zaprojektowany z myślą o przyszłej implementacji w nowoczesnych na tamten czas robotach. Główną ideą stojącą za utworzeniem tego języka opisu problemów było zauważenie, iż język STRIPS nie potrafi poradzić sobie z sytuacjami, gdy powstanie danego efektu jest niedeterministyczne. Dodatkowym aspektem, którego wprowadzenie znacznie poprawiło wydajność metodologii ADL względem STRIPS było wykorzystanie zasady **otwartego świata**, która mówi, iż rzeczy nieokreślone w świecie są uznawane jako *nieznane*, a nie jako fałszywe, jak to było w STRIPS (**zasada zamkniętego świata**). Dodatkowo, ostateczne cele w języku STRIPS mogły być jedynie przedstawione jako koniunkcje stanów (np. Dobra i Tania w kontekście restauracji), czyli jako cele bezwarunkowe. W nomenklaturze ADL cele można przedstawiać również przy pomocy alternatyw, to znaczy: nie wszystkie stany muszą być spełnione, mogły istnieć grupy stanów, w których spełnienie tylko jednego było wystarczające (np. Dobra i (Tania lub Blisko)).

Planning Domain Definition Language (w skrócie **PDDL**) było próbą ustandaryzowania języków wykorzystywanych do planowania przy pomocy komputerów silnie inspirowaną przez swoich poprzedników - STRIPS i ADL. Utworzeniem takiego standardu zajął się Drew McDermott wraz ze współpracownikami z **Yale University**. Podstawową ideą stojącą za PDDL jest całkowite odseparowanie metod rozwiązywania problemów od środowisk, w których się znajdują. Techniki rozwiązywania problemów związanych z logistiką (ustalanie najkrótszej drogi) powinny być również aplikowalne do rozwiązywania problemów związanych z produkcją (minimalizacja zużycia materiału). Pozostałe aspekty są porównywalne do języka STRIPS - istnieśnie warunków początkowych, zbioru operatorów oraz zbioru złożonego z celów do osiągnięcia.

1.2.5 Nowoczesne rozwiązania

Wraz z preżnym rozwojem informatyki, dziedzina planowania przy użyciu komputerów również zaopatryła się w nowy arsenał, mający na celu usprawnienie jak i ułatwienie generowania planów. Pojawiło się wiele innowacyjnych metod, które znacznie wykraczają poza ograniczenia, jakie nakłada język opisu świata STRIPS. Zaczęły powstawać nowe sposoby planowania takie jak planowanie preferencyjne, którego celem jest nie tylko utworzenie planu, ale również zapewnienie zachowania preferencji wskazanych obiektów świata. Często taki typ planowania może być wykorzystywany w sytuacjach, które obejmują ludzi, chociażby plan lekcji w szkole i preferencje nauczycieli odnośnie godzin pracy. Ponadto zaistniał diamentalny rozwój w programowaniu warunkowym, który może reagować na nieznane czynniki przedstawionego świata. Pozwala to na redukcję obszaru poszukiwań na kawałki, przez co sprowadza się do łatwiejszego przedstawiania złożonych problemów oraz uzyskiwania szybszych wyników przez algorytmy implementujące podejście warunkowe. Poza rozwojem, jeśli chodzi o opis dopuszczalnych akcji, dokonano również rozbudowy opisu stanów w świecie. Dla przykładu, **algorytmy online** charakteryzują się tym, iż są w stanie generować plany mimo braku posiadania całkowitej wiedzy o świecie w momencie rozpoczęcia pracy. Do algorytmu dane przekazywane są partiami. Na podstawie przesłanych informacji algorytm dokonuje odpowiednich decyzji i udziela odpowiedzi. Proces kontynuowany jest aż do momentu przetworzenia wszystkich wprowadzonych danych. Sztandarowym wykorzystaniem algorytmów online jest mechanizm przydzielania pamięci procesa do procesów w komputerze.

1.2.6 Miejsce GRAPHPLAN'u

Miedzy powstaniem metodologii STRIPS a jej rozszerzeniami w postaci ADL lub PDDL powstawały również inne podejścia, w tym silnie bazujący na grafach, właściwościach częściowego porządku i ich możliwościach, algorytm o wdzięcznej nazwie **GRAPHPLAN**.

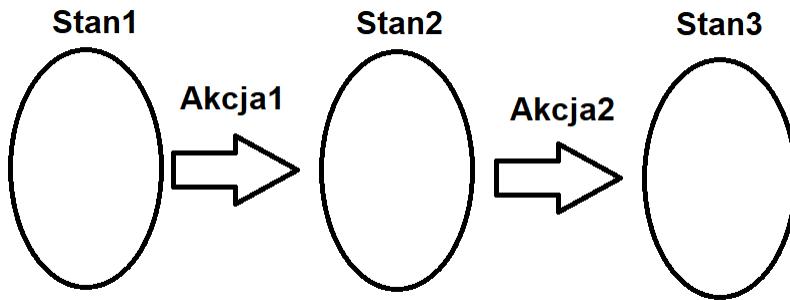


Rozdział 2

GRAPHPLAN

2.1 Wprowadzenie

GRAPHPLAN jest algorytmem do planowania akcji działającym w dziedzinie zdefiniowanej przez język STRIPS. Dodatkowo bazuje na paradygmacie, który autorzy algorytmu określają jako "graf planujący"^[3].



Rysunek 2.1: Najogólniejsza forma grafu planującego. Składa się on z węzłów, zwanych stanami oraz krawędzi zwanych akcjami. Docelowo poszczególne stany oraz akcje są parami różne, jednak mogą zajść sytuacje, gdy powtórzenie któregoś z komponentów będzie wymagane do uzyskania odpowiedniego celu.

Pierwotną ideę grafu planującego przedstawiono na obrazku 2.1. W trakcie dalszego omawiania metodologii GRAPHPLAN powyższy rysunek będzie pojawiała się ponownie z coraz to większym poziomem szczegółowości. Ze względu na fakt, iż Graphplan opiera się na języku STRIPS musi mieć jasno zdefiniowane: stan początkowy, akcje oraz cel, który należy uzyskać. Dzięki swojej strukturze Graphplan w swojej naturze podobny jest do programowania dynamicznego.

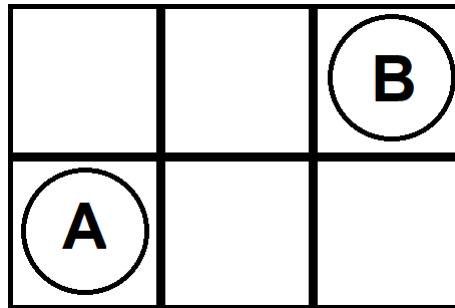
2.2 Warunki początkowe

Definicja 2.1 Zasada zamkniętego świata - Zasada, wedle której pojęcia, które nie są ściśle opisane w świecie są nieprawdziwe.

GRAPHPLAN, w odróżnieniu od człowieka, musi być w posiadaniu całej wiedzy o świecie, aby móc rozpoczęć działanie. Przez całą wiedzę o świecie rozumie się posiadanie informacji na temat każdego obiektu oraz jego stanu. Wiąże się to ściśle z faktem, iż GRAPHPLAN operuje zgodnie z definicją 2.1. Przed dalszą częścią pracy należy dokonać pewnego wyróżnienia. Słowo **stan** pojawia się w dwóch znaczeniach: stan jako pojedyncza informacja o obiekcie w świecie (Przykład: klocek B na stole numer 3) oraz stan, jako zbiór wszystkich takich informacji w danym momencie czasu. Z tego względu wprowadzono nowe pojęcie - "poziom stanów", które należy stosować jako oznaczenie wszystkich informacji o świecie w danym momencie.

Definicja 2.2 Poziom stanów - Zbiór informacji o stanach wszystkich obiektów w świecie w danej jednostce czasu t

Szczególnym poziomem stanów jest poziom oznaczany jako pierwszy i nazywany **Warunkami początkowymi**, którego poprawne zdefiniowanie jest kluczowym aspektem w kontekście uzyskania poprawnego wyniku przez algorytm. Analizując ponownie przykład 1.1 mylnym jest myśleć, iż jedyną informacją, jaką algorytm powinien posiadać o świecie jest pobyt klocka A na lewej platformie. Również istotną informacją jest brak klocka na platformie prawej, czyli informacja, że jest on *pusty*. Mimo poczucia nadmiarowości tej informacji, w dalszej części pracy wyjaśni się, dlaczego ta informacja jest niezbędna do uzyskania poprawnego wyniku. Przykład świata przedstawionego oraz skonstruowanego dla niego stanu początkowego:



Rysunek 2.2: Przykładowy moment startowy przyszłego planu. Za pomocą okręgów oznaczono roboty, natomiast poprzez kwadraty oznaczone są kafelki- miejsca, po których mogą poruszać się roboty.

Na powyższym przykładzie, zgodnie z ideą Graphplanu wyszczególniono 6 stanów początkowych. Dodatkowo należy doprecyzować pojęcie bycia robota na danym kafelku. Wykonano to przy pomocy dwuargumentowej relacji *na*, która jako pierwszy argument przyjmuje sygnaturę robota, a na drugim- numer kafelka. Na potrzeby przykładu ustalono, iż numerowanie odbywa się rzędami od lewej do prawej. Zgodnie z tymi ustaleniami pozycję robotów A i B można określić w następujący sposób: *na(A,4)* oraz *na(B,3)*. Również pustość kafelków należy sformalizować wprowadzając relację jednoargumentową o nazwie *pusty*, która przyjmuje jako argument numer pustego kafelka. Reasumując, zbiorem stanów początkowy dla analizowanego przykładu 2.2 jest:

$$\{pusty(1), pusty(2), na(B,3), na(A,4), pusty(5), pusty(6)\} \quad (2.1)$$

2.3 Akcje

Posiadając dobrze określony stan początkowy następnym krokiem będzie zdefiniowanie akcji. Zgodnie z 1.2 oraz wzmiiance o akcjach w planerze STRIPS, akcja musi składać się z trzech komponentów:

- czynności
- warunków zajścia
- efektów zajścia

Z tego powodu każdą z akcji traktuje się jako trójkę

$$A = (C, W, E) \quad (2.2)$$

gdzie każda z liter odpowiada pierwszej literze wyżej wymienionego pojęcia. W skład efektów wchodzą dwa pojęcia wprost z terminologii STRIPS- dodające i usuwające. Dzięki takiemu podziałowi łatwiejszym będzie zachowanie silnego podziału między przyczynami a efektami akcji. Jedyną czynnością, którą należy brać pod uwagę w ramach 2.2 jest czynność *ruch*, którą zdefiniowana jest jako trójargumentowa relacja:

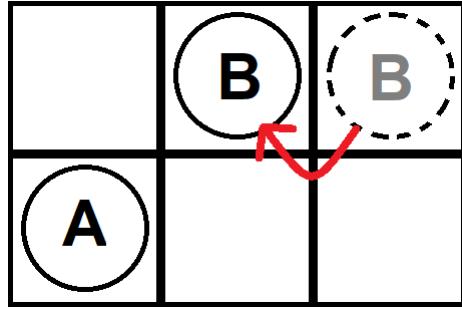
$$ruch(R, S, D) \quad (2.3)$$

gdzie R odpowiada robotowi, który musi się przemieścić z kafelka oznaczonego literą S (kafelek startowy) na kafelek oznaczony literą D (kafelek docelowy).

Następnymi składowymi są odpowiednio *Warunki* jak i *Efekty*. Warunki traktowane są jako zbiór wszystkich stanów, które muszą być prawdziwe w danej jednostce czasu. Jeśli choć jeden stan nie jest spełnialny, opisywana akcja nie może zostać wykonana w danym ruchu. Efekty natomiast zdefiniowane jako następującą parę:

$$E = (D, U) \quad (2.4)$$

gdzie D oznacza efekty dodające, a U- efekty usuwające.



Rysunek 2.3: Obrazowe przedstawienie ruchu robota B z kafelka 3 na kafelek 2

Niech rozpatrywaną akcją będzie przemieszczenie robota B z pozycji 3 na pozycję 2, przedstawiona na 2.3. Biorąc pod uwagę, iż stanem początkowym jest 2.1. Warunkami zajścia zdarzenia będą: $na(R, S)$ oraz $pusty(D)$. Dla efektów natomiast sytuacja wygląda następująco: efektami dodającymi są $na(R, D)$ oraz $pusty(S)$, które informują o tym, iż klocek wykonał ruch z kafelka S na kafelek D, a efektami usuwającymi $\sim na(R, S)$ oraz $\sim pusty(D)$, które informują o tym, iż kafelek S został zwolniony, robot nie znajduje się już na kafelku S oraz kafelek D już nie jest pusty. Przy pomocy matematycznego symbolu negacji wyrażono nieprawdziwość danego stanu. Ze względów estetycznych oraz ułatwiających analizowanie pracy negacje stanów w dalszej części wyrażono również przy pomocy polskiej partykuły przyczepowej **nie**. Dla przykładu pojęcia $\sim pusty(1)$ oraz $niepusty(1)$ z perspektywy wprowadzonej terminologii są tożsame.

Posiadając następującą wiedzę poniżej zdefiniowano jedyną akcję znajdująca się w prezentowanym przykładzie:

$$A = (ruch(R, S, D), \{na(R, S), pusty(D)\}, \{\{na(R, D), pusty(S)\}, \{\sim na(R, S), \sim pusty(D)\}\}) \quad (2.5)$$

Podstawiając za $R = B$, $S = 3$, a $D = 2$ otrzymano następującą akcję:

$$A = (ruch(B, 3, 2), \{na(B, 3), pusty(2)\}, \{\{na(B, 2), pusty(3)\}, \{\sim na(B, 3), \sim pusty(2)\}\}) \quad (2.6)$$

Analogicznie można zdefiniować ruch na kafelek numer 6, oraz dwa ruchy dla robota o sygnaturze A.

2.3.1 Typy akcji

Definicja 2.1 znajduje również swoje odzwierciedlenie w akcjach. Niech rozpatrywanym przykładem będzie wciąż przykład 2.2. W pierwszym kroku wykonano akcję $ruch(B, 3, 2)$. Z perspektywy człowieka jest to wystarczająca informacja, aby móc wydedukować, co we wskazanym etapie generowania planu dzieje się z robotem A. Otóż robot A w pierwszym ruchu zostaje na tym samym kafelku. Jednakże ze względu na zamkniętość świata algorytmu należy go również poinformować o tym, w jakim stanie po pierwszym kroku ma znajdować się robot A. Wykonywane jest to przy pomocy akcji, zwanych **akcjami podrzymującymi**.

Definicja 2.3 Akcja podrzymująca - Akcja, która przenosi stan obiektu w czasie t nienaruszonym do poziomu stanów w czasie $t + 1$

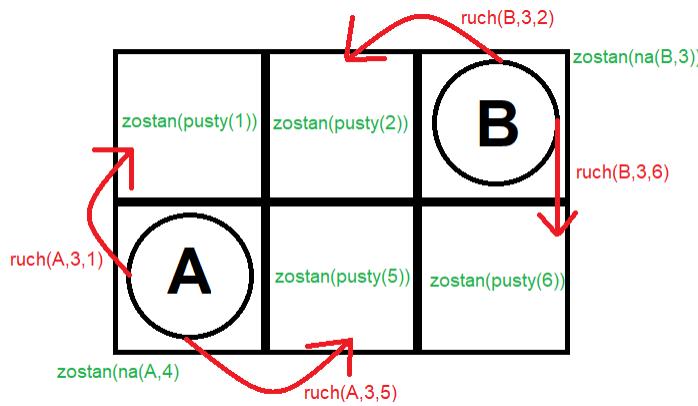
Akcjami podrzymującymi należy również informować świat o stanach kafelków, które nie brały udziału w akcji robota B. Przykładem takiego jest kafelek 1, który w stanie początkowym, jak i w stanie następnym ciągle zachowuje swój stan jako pusty. Akcje podrzymujące oznaczono słowem kluczowym **zostań**. Ponadto akcja typu *ruch*, które aktywnie zmienia stan świata w dalszej części pracy otrzyma miano **akcji aktywnej**.

Definicja 2.4 Akcja aktywa - Akcja, która zmienia stan obiektu między stanami w czasie t i $t + 1$.

Posiadając powyższy podział akcji poniżej przedstawiono pełen zbiór akcji w pierwszym kroku algorytmu. Wartym odnotowania jest, iż ze względów estetycznych podawawnie akcji podtrzymujących będzie często pomijane, jednakże nie można zapomnieć o ich występowaniu oraz o ich kluczowej roli w generowaniu precyzyjnego planu.

$$\text{Akcje} = \{\text{zostan}(pusty(1)), \text{zostan}(pusty(2)), \text{zostan}(pusty(5)), \text{zostan}(pusty(6)), \text{zostan}(na(B,3)), \\ \text{zostan}(na(A,4)), \text{ruch}(B,3,2), \text{ruch}(B,3,6), \text{ruch}(A,4,1), \text{ruch}(A,4,5)\}$$

Należy również nadmienić, iż podobnie jak w stanach, dla akcji wprowadza się pojęcie **poziomu akcji**, które funkcjonuje jako zbiór składający się ze wszystkich możliwych akcji do wykonania w danej jednostce czasu.



Rysunek 2.4: Obrazowe przedstawienie wszystkich akcji w pierwszym kroku algorytmu, akcje opisane przy użyciu czcionki o kolorze zielonym symbolizują akcje podtrzymujące, natomiast o kolorze czerwonym- akcje aktywne

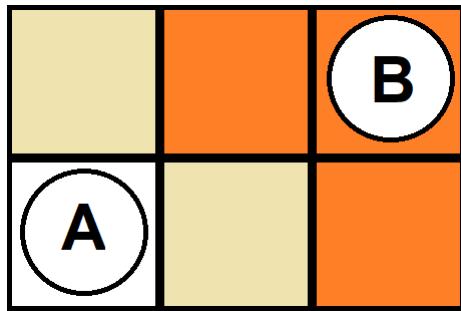
2.4 Definiowanie świata

Zdecydowanie najtrudniejszym aspektem modelowania świata jest dokładne przedstawienie wszystkich zależności, jakie algorytm musi znać, aby mógł bezbłędnie wnioskować w prezentowanej przestrzeni. Analizując przykład 2.2 dokonano przedstawienia schematu generowania poziomu stanów początkowych oraz poziomu akcji. Zadając pytanie algorytmowi *Jakie działania należy podjąć, aby jak najszybciej przesunąć robota A z klocka 4 na klocka 6* algorytm odpowie w następujący sposób: $ruch(A,4,6)$, jednakże przyglądając się uważnie przedstawionemu światu zauważono, iż taki plan byłby realny jedynie, gdyby akcja *ruch* oznaczała latańie- wtedy faktem jest, iż istnieje możliwość pominięcia kafelka 5 podczas przemieszczenia. Niespójność ta pojawiła się z faktu, iż w żadnym momencie nie określono w jaki sposób dokładnie funkcjonuje czynność oznaczona jako *ruch*. Wedle definicji 2.5 wszystkie warunki, aby robot mógł przemieścić się z kafelka 4 na 6 zostały spełnione.

Naprawić ten problem można przy pomocy uściślenia wykonywanej czynności. Do tego potrzebne będzie wprowadzenie relacji *sąsiad*, która przyjmuje dwa argumenty- numery kafelków, które ze sobą sąsiadują. Zakładając, iż sąsiadujące kafelki to takie, które mają wspólną ścianę, sąsiadami kafelka 1 są kafelki: 2 oraz 4. Dla pozostałych przestrzeni dokonano analogicznego wygenerowania listy sąsiadów. Dzięki tej operacji, akcję *ruch* zdefiniowano w następujący sposób:

$$ruch(R,S,D):-\text{sasiad}(S,D) \quad (2.7)$$

Czynność ruchu w powyższym przypadku została określona zgodnie z semantyką języka programowania **PROLOG**. Należy to rozumieć w następujący sposób: po lewej stronie znaku $:$ - znajduje się *konkluzja*, natomiast po prawej- *przesłanka*. Naturalnym odczytem przedstawionej sytuacji będzie zdanie "Jeśli kafelki S i D są sąsiadami to możliwym do wykonania jest ruch między tymi kafelkami"



Rysunek 2.5: Graficzne przedstawienie relacji sąsiedztwa dla kafelka numer 4. Kolorem kremowym oznaczono miejsca, z którymi sąsiaduje, natomiast pomarańczowym- te, z którymi nie sąsiaduje. Oznacza to, iż do kafelków pomarańczowych nie można dostać się wykonując jeden ruch

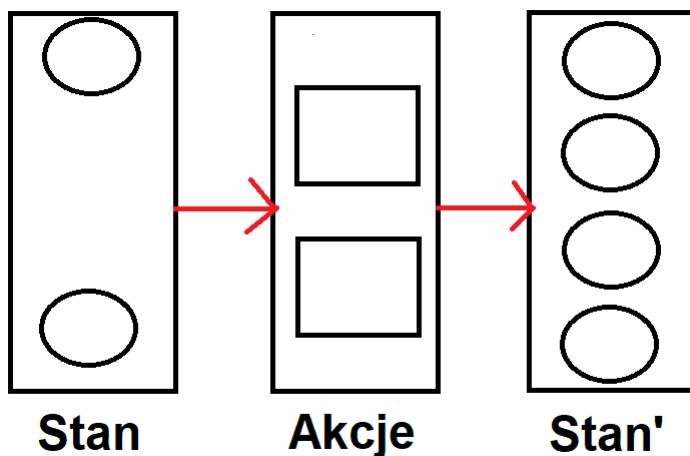
Taka definicja ruchu pozwala w skuteczny sposób oddać intuicję, która wynika z poglądowego obrazka przedstawiającego rozpatrywany świat. Niewystarczającym jest określenie relacji $sasiad(1,2)$, gdyż wedle tego faktu kafelek pierwszy sąsiaduje z kafelkiem drugim, jednakże nie oznacza to, iż kafelek drugi sąsiaduje z kafelkiem pierwszym. Wedle tej informacji dla każdej pary kafelków A i B do zbioru sąsiadów należy dodać dwie relacje: $sasiad(A,B)$ oraz $sasiad(B,A)$. Inna sytuacja byłaby, gdyby ruch był możliwy jedynie w jedną stronę, jednakże tutaj taki stan rzeczy nie występuje.

Oprócz definicji sąsiedztwa należy również zabezpieczyć się przed sytuacją, gdy dwa roboty będą chcięły w tym samym momencie znaleźć się na tym samym kafelku. Dodatkowo kafelek A nie może znajdować się w dwóch stanach jednocześnie: $pusty(A)$ oraz $na(R,A)$, gdzie R oznacza dowolnego robota. Wymodelowanie tych ograniczeń jest żmudnym, lecz istotnym zadaniem implementując przedstawianą metodologię, które zostanie poruszone w ramach przedstawiania pojęcia "wzajemnego wykluczania".

Odpowiednie wymodelowanie świata wedle wzorców języka STRIPS może wydawać się trudnym zajęciem, jednakże przebrnawszy przez ten etap algorytm jest zwarty i gotowy do generowania planów dla wprowadzonych celów.

2.5 Warstwy grafu

Składowe grafu planującego można podzielić na dwa typy: jeden, wyszczególniony jako poziomy stanów oraz drugi- poziomy akcji. Aby lepiej uwidoczyć zależność poziomu akcji od warunków początkowych oraz zależność kolejnego poziomu stanów od efektów akcji graf planujący z 2.1 ulegnie lekkiej modyfikacji.

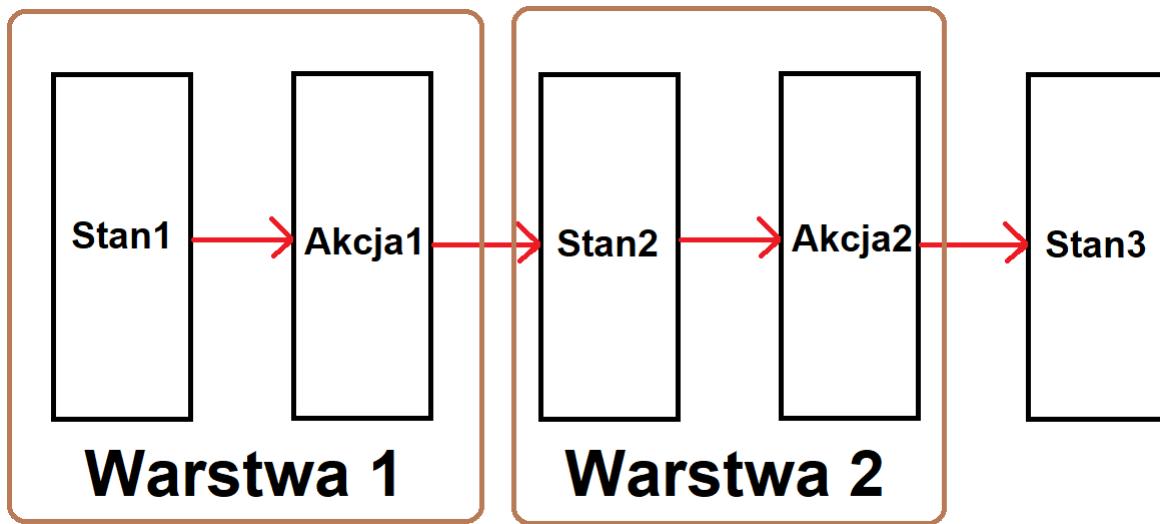


Rysunek 2.6: Modyfikacja do grafu planującego wprowadzona w poprzednich podrozdziałach. Wprowadzono zależności akcji od stanu poprzedniego, oraz stanu następnego od akcji.

Następnym krokiem będzie wprowadzenie definicji **Warstwy grafu**

Definicja 2.5 *Warstwą - grafu jest połączenie poziomu stanów oraz wynikającego z niego poziomu akcji*

Przez i -tą warstwę grafu oznaczono stan świata w i -tym momencie czasu. Ze względu na charakterystykę planera czas traktowany jest w sposób dyskretny- każda ze zdefiniowanych akcji zajmuje zawsze tyle samo czasu oraz zawsze kończy się tym samym efektem. Poziomy stanów jak i poziomy akcji należące do tej samej i -tej warstwy nazwano poprzez dodanie do ich nazwy numeru, odpowiadającego obecnej iteracji algorytmu



Rysunek 2.7: Przedstawienie sposobu wyznaczania warstw w grafie planującym. Poziom stanów oraz akcji wchodzący w skład danej warstwy wyróżniony jest poprzez konkatenację nazwy oraz liczby symbolizującej numer warstwy.

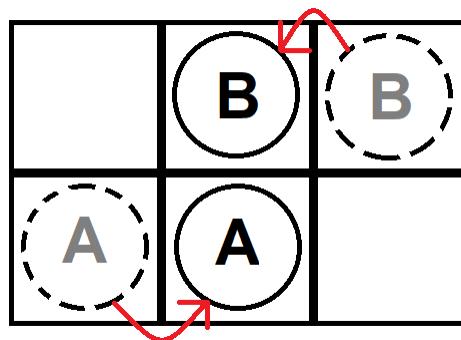
Dzięki wprowadzeniu definicji warstwy po wygenerowaniu planu natychmiast wiadomo ile kroków należy wykonać, a co za tym idzie- ile czasu należy poświęcić, aby osiągnąć ustalony cel.

2.6 Równoległość

W poprzednich paragrafach na pojedynczym poziomie akcji rozpatrywano co najwyżej jedną akcję aktywną, jednakże główną siłę GRAPHPLANU, jest możliwość jego reprezentacji jako częściowego porządku. To pozwala w prosty sposób wprowadzić pojęcie równoległości między akcjami.

Definicja 2.6 *Dwie (lub więcej) akcje mogą zostać wykonane równolegle, gdy po ich wykonaniu w tej samej warstwie i , warstwa $i + 1$ nie będzie zawierała w sobie żadnych sprzeczności.*

Przyglądając się rysunkowi 2.2 od razu należy zauważyc, iż w stanie początkowym ruch robota A nie wpływa w żaden sposób na otoczenie robota B. Z tego względu dwie przykładowe akcje $ruch(A,4,5)$ oraz $ruch(B,3,2)$ należy wykonać w tej samej jednostce czasu, czyli w pierwszym poziomie akcji.



Rysunek 2.8: Przykład możliwości zastosowania równoległości w planowaniu działania. Należy zauważyć, iż ruchy robotów w żaden sposób ze sobą nie kolidują.

Po wprowadzeniu definicji równoległości, również definicja kroku musi ulec ewolucji.

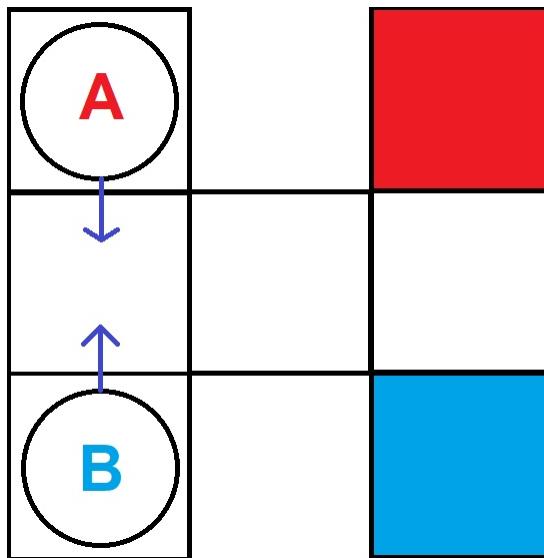
Definicja 2.7 *Krokiem algorytmu jest zbiór wszystkich możliwych do realizacji akcji w danej warstwie.*

Dzięki naniesionej poprawce możliwym jest wykonanie więcej niż jednej akcji między dwoma poziomami stanów. W tym miejscu należy zauważyć, iż poprzednie analizy zawierały spore uproszczenie, gdyż akcje podtrzymujące również należą do kroku algorytmu, co zostało zaprezentowane w trakcie kreowania pierwszej warstwy akcji dla rozpatrywanego przykładu.

Istnieją przypadki, gdy podmiot realizujący plan nie jest w stanie wykorzystać benefitów płynących z możliwości dokonywania akcji równolegle, na przykład ze względu na ograniczone zasoby. GRAPHPLAN również rozwiązuje ten problem, pozwalając przekształcać swój częściowy porządek na porządek liniowy w dość dowolny sposób. Otóż akcje z danej warstwy mogą być wykonywane w dowolnej kolejności, gdyż w żaden sposób ze sobą nie kolidują. Z perspektywy algorytmu najważniejszym jest, aby stan świata i i $i+1$ był zgodny z informacjami jakie posiada o świecie. Zgodnie z przykładem 2.9 widać, iż to, czy akcja $ruch(A,4,5)$ zostanie wykonana przed akcją $ruch(B,3,2)$ lub to, czy akcja $ruch(B,3,2)$ odbędzie się przed $ruch(A,4,5)$ nie ma większego znaczenia- efekt końcowy jest identyczny.

2.6.1 Wzajemne wykluczanie

Twórcy algorytmu wprowadzając równoległość, zdawali sobie sprawę z mocy tego podejścia. Dzięki temu ogromna liczba planów ulega redukcji jeśli chodzi o wymagany czas wykonania. Jednakże równoległość wprowadza kolejny istotny problem w prezentowanym świecie, a mianowicie- co, gdy wykonanie dwóch akcji będzie wprowadzało sprzeczność w następnym poziomie świata? Sytuacja ta została przedstawiona na poniższym rysunku



Rysunek 2.9: Przykład świata, w którym wprowadzenie równoległości dla pierwszej warstwy algorytmu jest niemożliwe. Dwa roboty próbują przejść na ten sam kafelek w tej samej jednostce czasu, co z perspektywy kafelka powoduje konflikt. Odpowiednio kolorami: czerwonym i niebieskim oznaczono roboty oraz kafelki, które są ich celem.

Zmusiło to twórców algorytmu do wprowadzenia pojęcia **relacji wykluczania**. O akcjach wzajemnie się wykluczających (ang. actions mutually exclusive, **mutex**) wspomiano w ramach definiowana świata. Pojęcie ów można określić w następujący sposób:

Definicja 2.8 *Relacją wzajemnie wykluczającą się - jest relacją między akcjami(stanami), która informuje o tym, iż nie istnieje plan taki, aby dwie wybrane akcje(stany) mogły być prawdziwe w tej samej jednostce czasu t*

Przykładem stanów wykluczających jest para: *pusty(1)* oraz *niepusty(1)*. Kafelek nie może być pusty jak i niepusty jednocześnie. Przykład dwóch akcji wykluczających przedstawiono na rysunku 2.9. Należy zauważyć, iż wzajemne wykluczanie się jest rozpatrywane warstwowo. Kolejnym naturalnym krokiem jest ustalenie, kiedy akcje oraz stany są ze sobą w relacji wykluczającej.

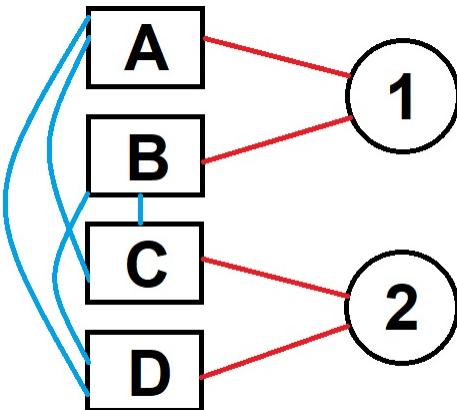
Wykluczanie się stanów

Dwa stany są ze sobą w relacji wzajemnie wykluczającej w dwóch następujących przypadkach:

1. Negacja- przypadek, w którym jeden ze stanów jest negacją drugiego
2. Niespójne powstanie- wszystkie akcje z poprzedniej warstwy, które prowadzą do utworzenia dwóch stanów są ze sobą parami w relacji wykluczającej

Ze względu na naturalność pojęcia negacji złożonym jest wprowadzenie większego przykładu niż przedstawienie, iż dla każdego obiektu istnieje para stanów znajdujących się w relacji wykluczającej. Niech za przykład posłuży pustota kafelka. Kafelek nie może być jednocześnie pusty, jak i niepusty, co sprowadza się, iż stan *pusty(kafelek)* jak i \sim *pusty(kafelek)* są niemożliwe do zawarcia w jednej warstwie planu.

Dla kontrastu przedstawienie przypadku, w którym zachodzi **niespójne powstawanie** jest trudniejsze i wymaga użycia przykładowej ilustracji:



Rysunek 2.10: Urywek planu przedstawiający sytuację, w której dwa stany powstają w sposób niespójny. Zgodnie z założeniami akcje oznaczone są przy pomocy prostokątów, stany - okręgów, między akcjami a stanami czerwone linie symbolizują, które stany są efektami których akcji, natomiast linie niebieskie między akcjami symbolizują powstające między nimi *mutex*.

Zgodnie z przykładem 2.10 należy zauważyć, iż stany 1 oraz 2 nie mogą znajdować się jednocześnie w planie ze względu na to, iż każda z akcji, która generuje stan 1 (B,C) znajduje się w relacji wzajemnie wykluczającej z każdą akcją, która generuje stan 2 (C,D). Oznacza to, iż ów dwa stany powstają w sposób **niespójny**.

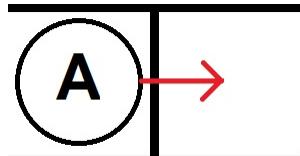
Z reguły ten typ wykluczeń występuje po większej liczbie kroków dla bardziej rozbudowanych światów jak i planów.

Wykluczanie się akcji

Dwie akcje mogą być ze sobą w relacji wzajemnie wykluczającej w trzech następujących przypadkach:

1. Niespójny efekt- przypadek, w którym zbiór efektów jednej z akcji jest negowany przez zbiór efektu drugiej
2. Przeszkadzanie - przypadek, w którym jedna z akcji usuwa warunki zajścia akcji drugiej
3. Konkurencyjne potrzeby- przypadek, w którym warunki zajścia akcji są ze sobą w relacji wykluczającej.

Poniższe przykłady w obrazowy sposób przedstawiają każdy z wyżej wymienionych przypadków:



Rysunek 2.11: Przykład wykluczania się akcji aktywnych z podtrzymującymi

Na pierwszy rzut oka wydawać by się mogło, iż niemożliwym jest wygenerowanie dwóch akcji znajdujących się w relacji wykluczania dla przykładu 2.11, jednakże istnieją dwie takie pary:

$$\text{zostan}(\text{na}(A,\text{lewy})), \text{ruch}(A,\text{lewy},\text{prawy}) \quad (2.8)$$

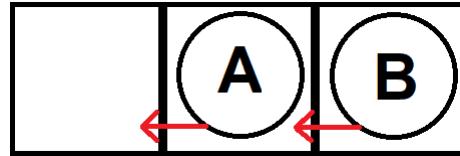
oraz

$$\text{zostan}(\text{pusty}(\text{prawy})), \text{ruch}(A,\text{lewy},\text{prawy}) \quad (2.9)$$

Wynika to z faktu, iż zbiór efektów jednej akcji z powyższych par jest negowany przez drugi. Zbiorem efektów akcji *zostan(na(A,lewy))* jest *na(A,lewy)*, jednakże zbiorem efektów *ruch(A,lewy,prawy)* jest ciut większy zbiór:

$$\text{na}(A,\text{prawy}), \sim \text{na}(A,\text{lewy}), \text{pusty}(\text{lewy}), \sim \text{pusty}(\text{prawy}) \quad (2.10)$$

Widać, iż efekt $\sim na(A, lewy)$ jest negacją efektu $na(A, lewy)$. Akcje podtrzymujące i aktywne względem tego samego obiektu zawsze sobą ze sobą w relacji wykluczającej ze względu na **niespójny efekt**.



Rysunek 2.12: Przykład świata, w którym każdy z robotów próbuje przesunąć się na sąsiadujący z lewej strony klocek. Uwidoczniona sytuacja jest przykładem wykluczających się akcji

Idea przedstawiona na rysunku 2.12 jest ciekawym przykładem, pozwalającym zrozumieć dokładniej definicję słowa **równoległość** odnośnie planów generowanych przez GRAPHPLAN. Gdyby roboty, poruszając się z tą samą szybkością, ruszyły w tym samym momencie to wykonanie ów sekwencji akcji byłoby w zupełności możliwe, jednakże w sekcji 2.6 wspomniano, iż każdy z wygenerowanych planów musi być możliwy do przedstawienia w postaci porządku liniowego, czyli plan równoległy może zostać przerobiony w dowolny sposób na plan, w którym każda z akcji wykonywana jest w góry ustalonej kolejności. Zgodnie z omawianym przykładem sprowadzenie ów planu do dowolnej postaci liniowej jest niemożliwe, gdyż istnieje konfiguracja, w której to najpierw robot B miałby wykonać ruch, jednakże jest to niemożliwe ze względu na obecność robota A na kafelku będącym jego celem podróży. Dodatkowe wprowadzenie ograniczeń na konwersję planów równoległych na liniowe jest bardziej skompilowanym zagadnieniem, którego omówienie nastąpi w sekcji odpowiedzialnej za możliwe rozwinięcia algorytmu.

Z powyższego opisu wynika, iż omawiane relacje ruchu są ze sobą w relacji wzajemnego wykluczającej z powodu **przeszkadzania**. Kloczek A do wykonania ruchu potrzebuje znajdować się na kafelku środkowym, co oznacza, iż kafelek środkowy musi być niepusty, jednakże kloczek B potrzebuje, aby ów klocek był pusty. Relacja wykluczająca między tymi stanami bezpośrednio generuje wykluczanie się tych akcji.

Ponadto, omawiane akcje w ilustracji 2.12 znajdują się w relacji wykluczającej z powodu **konkurencyjnych potrzeb**. Robot B przy przesunięciu **na** środkowy kafelek wymaga, aby on był pusty, natomiast robot A przy przesunięciu **z** środkowego klocka musi się na nim znajdować, czyli kafelek musi być niepusty. Stany pusty i niepusty względem kafelka są w oczywistej relacji wykluczającej co automatycznie generuje wykluczenie się wspomnianej pary akcji. Przykład ów przedstawia, iż jedna para akcji może być ze sobą w relacji wykluczającej z kilku powodów, jednakże algorytm rozpatrując parę akcji podchodzi do tego w sposób binarny, co oznacza, iż patrzy jedynie czy znajdują się w relacji wykluczającej czy nie, nie interesuje go liczba sposobów, na które ów relację można utworzyć.

Należy zauważać, iż dzięki wprowadzeniu relacji wzajemnego wykluczania pozbyto się porównań wielu akcji, których zajście jest niemożliwe w tej samej warstwie grafu. Koszt jaki został poniesiony ze względu na zapamiętywanie dodatkowych informacji o relacjach między akcjami jest często zaniedbywalny, ze względu na ogrom benefitów w postaci mniejszej liczby sprawdzeń, a co za tym idzie - szybszego działanie algorytmu.

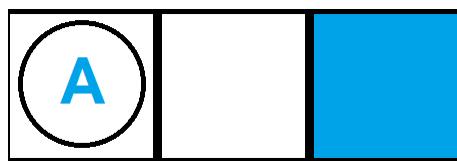
2.7 Wyszukiwanie planu

Zdefiniowawszy wszystkie niezbędne elementy planera należy wskazać, w jaki sposób przy posiadaniu całej wiedzy wygenerowanej przez graf planujący algorytm tworzy odpowiedni plan. Wykonywane jest to w następujący sposób:

1. Rozpoczynając od stanu początkowego, zgodnie z opisany w poprzednich sekcjach metodami, odbywa się generowanie kolejnych warstw grafu, biorąc pod uwagę informację o **mutexach** między akcjami oraz stanami
2. Dla każdej nowo utworzonej warstwy i dokonywane jest sprawdzenie, czy wszystkie założenia z celu nie znajdują się w ów poziomie stanów. Jeśli odpowiedź jest negatywna, odbywa się dalsze generowanie planu zgodnie z 1. Jednak, gdy wszystkie stany celu znajdują się na poziomie i dochodzi do generowania planu.

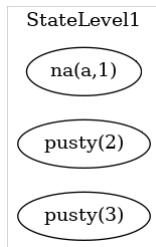
3. Dla każdego stanu z celów na poziomie i dochodzi do wybrania akcji, dzięki której został on wygenerowany. Ów operacja dokonywana jest dla każdego stanu. Jeżeli dobrane akcje są ze sobą w relacji wykluczającej, należy spróbować innej kombinacji akcji. Jeśli wszystkie dobory akcji zawiodą należy wygenerować kolejną warstwę grafu i ponownie, w warstwie $i + 1$, rozpoczęć cały proces
4. Jeśli jednak istnieje dobór akcji taki, że nie występuje między nimi relacja wykluczania, należy dla każdego stanu znajdującego się w zbiorze warunków zajścia wspomnianych akcji wykonać procedurę z kroku 3.
5. Jeśli dojdzie do niepowodzenia na którymkolwiek z etapów powrotu do stany wyjściowego algorytm podejmuje próbę odpowiedniego dobrania akcji na ostatnio sprawdzonym poziomie. Brak niepowodzenia na ścieżce powrotu od warstwy i do warunków początkowych świadczy o tym, iż istnieje sekwencja akcji pozwalająca otrzymać stany zdefiniowane z celu w określonym przez stany początkowe świecie, co oznacza, iż jest możliwym utworzenie odpowiedniego **planu**.

Aby precyzyjnie przedstawić działanie algorytmu w następnej części programu przeprowadzono rozbudowaną analizę konkretnego przykładu:



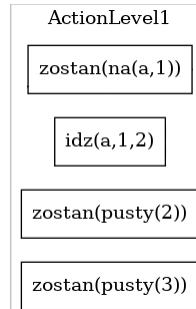
Rysunek 2.13: Sytuacja początkowa świata, dla którego odbędzie się przykładowe generowanie planu. Kolorem niebieskim zaznaczono kafelek docelowy robota

2.13 przedstawia świat, w którym robot A z kafelka 1 próbuje przedostać się do kafelka 3 (Kafelki ponownie numerowane są od lewej strony do prawej).



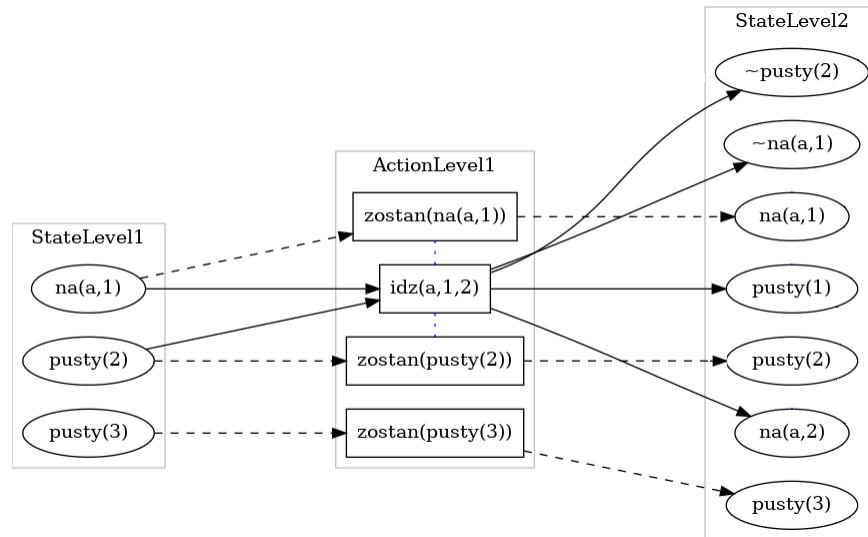
Rysunek 2.14: Warunki początkowe omawianego świata. Obrazki ułatwiające analizę przykładu w całości zostały wykonane przez oprogramowanie utworzone na rzecz pracy.

Ze względu na fakt, iż w języku programowania **PROLOG**, w którym implementowany jest algorytm, zmienne są oznaczane przy pomocy wielkiej litery, a stałe przy pomocy małej, należy nie przywiązywać większej wagi do wielkości litery prezentowanej na przytoczonych rysunkach. Zwyczajowo w opisie robot będzie określany przy pomocy wielkiej litery (np. **A**) natomiast w wygenerowanych przez program grafach przy pomocy małej **a**. W powyższy sposób zaprezentowano stan początkowy świata. Wszystkie definicje relacji takich jak **na**, **idz**, **zostan**, czy **pusty** wprowadzono w podrozdziałach 2.2 i 2.3. W wygenerowanych przez oprogramowanie grafach **poziomy stanów** określone są poprzez swój angielski odpowiednich **StateLevel**. Podobnie z **poziom akcji** i **ActionLevel**.



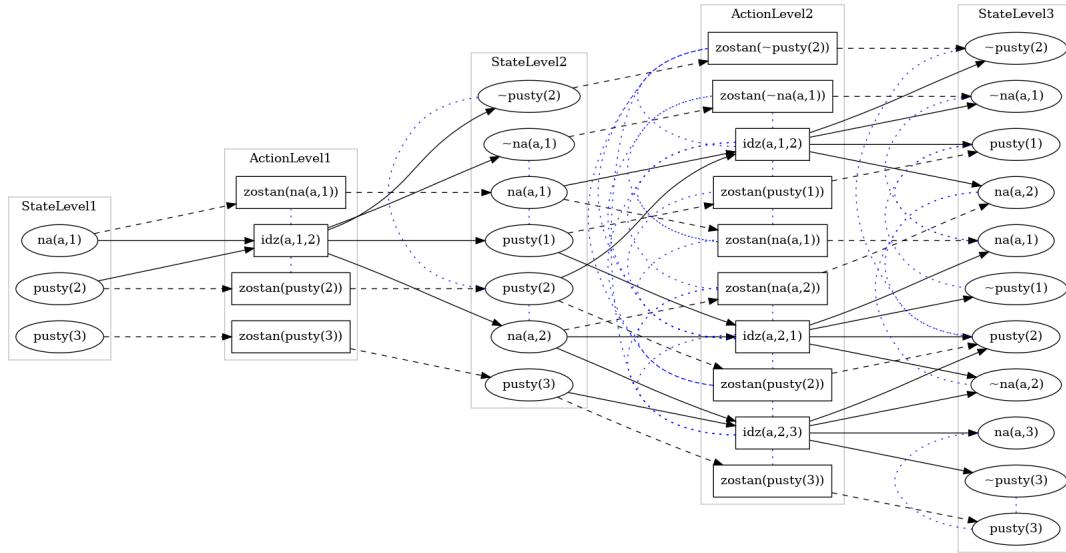
Rysunek 2.15: Wygenerowane akcje na podstawie warunków początkowych

Następnie przystąpiono do wygenerowania wszystkich możliwych akcji. Postąpiono zgodnie ze wskazówkami z podrozdziału 2.3, dodawszy niezbędne krawędzie wynikające z warunków, jak i efektów każdej z akcji oraz mutex (oznaczone przerywanymi, niebieskimi liniami).



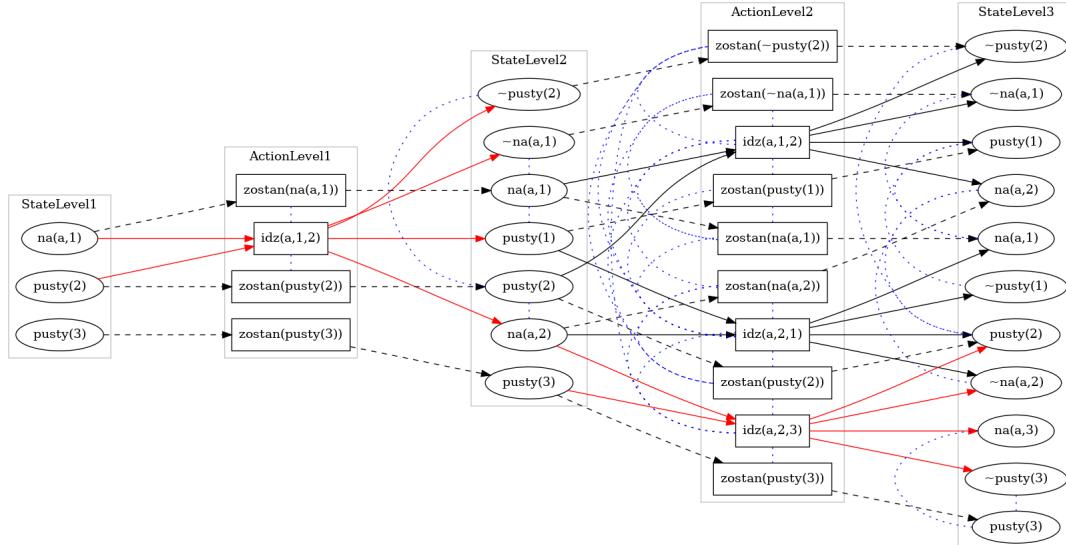
Rysunek 2.16: Wygenerowany drugi stan grafu bezpośrednio wynikający z pierwszego poziomu akcji

Przy pomocy efektów akcji automatycznie wykreowano następny poziom stanów. Zgodnie z schematem planowania należy sprawdzić, czy oczekiwany cel $na(A,3)$ znajduje się w zbiorze stanów poziomu drugiego. Po szybkiej analizie okazuje się, iż cel nie znajduje się na wskazanym poziomie, więc należy dokonać rozszerzenia grafu planującego o kolejny poziom. Następuje to w sposób analogiczny.



Rysunek 2.17: Przykład kolejnej iteracji algorytmu

Po wygenerowaniu kolejnej warstwy algorytmu oraz zaznaczeniu wszystkich mutexów zauważono, iż stan $na(A,3)$ znajduje się w zbiorze stanów **StateLevel3**. Następnym krokiem jest sprawdzenie, czy taki zbiór akcji, który doprowadzi nas do stanu $i-1$ bez wprowadzania żadnych sprzeczności. Okazuje się, iż istnieje taka para akcji: $idz(A,2,3)$ oraz $zostan(pusty(1))$. Zgodnie z schematem generowania planu należy zejść iteracyjnie aż do warunków początkowych aby uzyskać poprawny plan. W omawianym przykładzie łatwo zauważać, iż przy pomocy akcji $idz(A,1,2)$ oraz $pusty(3)$ udało się z powrotem otrzymać stan początkowy. Ze względu na fakt, iż człowiek jest w stanie sprawnie wydedukować akcje podtrzymujące na podstawie akcji aktywnej, dla każdego poziomu przy graficznym przedstawieniu planu (w formie opisowej i graficznej) z reguły akcje podtrzymujące będą pomijane.



Rysunek 2.18: Graf planujący z zaznaczonymi akcjami aktywnymi prowadzącymi do uzyskania wskazanego celu

Powyższy rysunek przy pomocy czerwonych strzałek przedstawia akcję aktywną, wraz z jej bezpośrednimi warunkami zajścia oraz efektami, które prowadzą do uzyskania planu. Zgodnie z intuicją, najbardziej

optymalny plan przeniesienia robota A z kafelka 1 na kafelek 3 to:

$$[[idz(A,1,2)],[idz(A,2,3)]]$$
 (2.11)

Ze względu na fakt, iż na danym poziomie może odbywać się więcej niż jedna akcja, każdy krok algorytmu oznaczany jest za pomocą **listy**, którą należy utożsamiać z matematycznym zbiorem.

W ten oto sposób wygenerowano poprawny plan stosując algorytm **GRAPHPLAN**.

2.8 Własności GRAPHPLAN'u

Działanie GRAPHPLAN'u można przedstawić również jako swego rodzaju iteratywne przeszukiwanie wszerz. Dla każdego poziomu generowane są wszystkie możliwe stany, by następnie móc sprawdzić, czy wyznaczone cele znajdują się w utworzonym zbiorze. Jeśli rzeczone cele nie znajdują się, algorytm wykonuje kolejną iterację, poszerzając swoją wiedzę o świecie. Powyższe przeszukiwanie wykonuje się aż do momentu, gdy uda się utworzyć satysfakcyjny plan. Dzięki tej własności łatwo pokazać, iż:

Twierdzenie 2.1 *Każdy wyprodukowany plan przez algorytm GRAPHPLAN jest planem legalnym, czyli spełnialnym dla zadanego warunków początkowych, akcji oraz celów. Ponadto, jeśli dla opisanego świata istnieje plan GRAPHPLAN zawsze go znajdzie.*

oraz

Twierdzenie 2.2 *Algorytm zawsze zwraca najkrótszy plan, czyli plan optymalny.*

Dowód. W pierwszym kroku algorytm generuje nowy poziom stanów. Następnie sprawdza, czy cele znajdują się wśród stanów. Jeśli odpowiedź jest twierdząca, dobiera zbiór akcji, dzięki którym udało się spełnić postawione wymagania. Jeśli odpowiedź jest przecząca, algorytm generuje kolejną warstwę. Niech warstwa i będzie warstwa, w której nie znajdują się wszystkie stany ze zbioru determinowanego przez zdefiniowane cele. W tym przypadku dla $i + 1$ warstwy algorytm próbuje wygenerować plan cofając się do stanu i . Jeśli uda się odpowiednio dobrą akcję cofać się jeszcze dalej aż do stanu początkowego i kończy działanie, jako wynik zwracając plan. Powyższy opis determinuje optymalność planu. ♦

Ponadto należy się również przyjrzeć mechanizmom kreowania kolejnych stanów dla poszczególnych warstw. Algorytm robi to przyrostowo. Wszystkie istniejące już stany przenoszone są przy pomocy akcji podtrzymujących do następnej warstwy, natomiast akcje aktywne produkują nowe efekty- dodające bądź usuwające. Mimo iż akcja z perspektywy człowieka jest **usuwająca**, z perspektywy komputera jest kolejną informacją, która również zostaje dodana do zbioru stanów.

Korzystając z powyższej własności sformułowano następujący wniosek:

Twierdzenie 2.3 *Liczba stanów w warstwie $i+1$ jest zawsze większa bądź równa liczbie stanów w warstwie i*

Skutkiem tego twierdzenia są dobrze uwidocznione w przykładach grafów planujących takich jak 2.19. Twierdzenie 2.3 generuje ciekawy wniosek. Niech oznaczenie $WSL(i)$ oznacza liczbę stanów wchodzących w skład i -tej warstwy.

Lemat 2.1 $\exists k \in \mathbb{N} \forall i > k, i \in \mathbb{N} : WSL(i+1) = WSL(i)$

Co sprowadza się do tego, iż od pewnego momentu liczba stanów dla każdej kolejnej warstwy jest stała.

Dowód. Zgodnie z 2.3 wiadomość jest, iż zawsze spełniona jest zależność $WSL(i+1) > WSL(i)$. Należy pokazać, iż od pewnego momentu liczby stanów w warstwach dążą do stałej liczby. Ze względu na strukturę języka STRIPS istnieje skończona liczba akcji. Wówczas nie istnieją akcje niedeterministyczne, bądź jakiekolwiek niespodziewane efekty uboczne przeprowadzonych akcji. Algorytm również rozpoczęta pracę z góry znaną liczbą obiektów, a każdy z obiektów może znajdować się w skończonej liczbie stanów takich jak *na* czy *pusty*. Algorytm rozszerza kolejne poziomy stanów przy pomocy akcji aktywnych, jednakże na podstawie skończonej liczby obiektów oraz skończonych stanów, w których obiekty mogą się znajdować algorytm, dla każdego z obiektów, może wywnioskować jedynie skończoną liczbę akcji, które może wykonać. Ostatecznie sprowadza się to do sytuacji, w której ze względu na skończoność wszystkich określonych dziedzin, i -ty poziom

stanów algorytmu zawiera wszystkie możliwe stany dla wszystkich możliwych obiektów. Od tego momentu mimo przeprowadzania kolejnych akcji podtrzymujących bądź aktywnych nie powstają już kolejne stany, gdyż zbiór efektów każdej z akcji ma już swoje odwzorowanie w poprzedniej warstwie. Stąd wniosek, iż istnieje moment krytyczny, dla którego dochodzi do stabilizacji liczby stanów w danej warstwie. ♦

Definicja 2.9 *Spłaszczeniem algorytmu jest sytuacja, w której wygenerowanie kolejnego poziomu stanów nie prowadzi do uzyskania nowych informacji o świecie.*

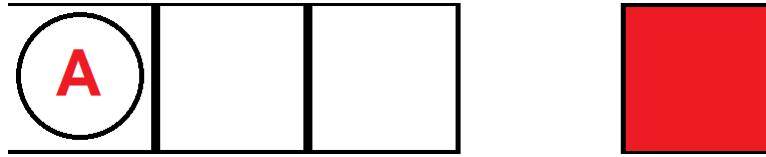
Powyższy lemat jest kluczowym w kontekście rozwiązywania tak zwanego *Problemu stopu* dla algorytmu GRAPHPLAN.

2.8.1 Złożoność obliczeniowa

Twierdzenie 2.4 *Niech problem do rozwiązywania przy pomocy planu składa się z n obiektów, p stanów w warstwie początkowej, m operatorów przedstawionych zgodnie z zasadami języka STRIPS, posiadających stałą liczbę parametrów. Dodatkowo, niech l będzie długością najdłuższej listy efektów dodających dla zbioru akcji. Wtedy rozmiar t poziomowego grafu planującego wygenerowanego przez algorytm, jak i czas jego generacji jest wielomianowy w zależności od zmiennych n, m, p, l i t .*

Co jest znacznym usprawnieniem względem poprzednich metod. Dowód tego twierdzenia można znaleźć w oryginalnej pracy autorów [3].

2.8.2 Problem stopu



Rysunek 2.19: Przykład świata, w którym osiągnięcie celu, którym jest kafelek oznaczony kolorem takim jak kolor użytej czcionki do oznaczenia sygnatury robota, jest niemożliwe ze względu na absencję jednego z kafelków, będących łącznikiem, między robotem a jego celem.

Wcześniejsze przykłady rozpatrywały jedynie sytuację, w której algorytm zawsze znajdował plan, gdyż przedstawione światy były spreparowane w taki sposób, aby rzeczywisty plan zawsze istniał. Jednakże nie zawsze musi tak być, więc należy zabezpieczyć algorytm również i przed taką ewentualnością. Łatwo zauważać, iż obecna struktura planu nie została utworzona z myślą o takiej sytuacji- algorytm będzie generował coraz to nowsze warstwy bezskutecznie próbując uzyskać plan dla zdefiniowanego celu. Ostatecznie dojdzie do całkowitego zapętlenia pracy algorytmu. Z tego powodu pierwszym pomysłem na uzbrojenie algorytmu w mechanizm detekcji nieskończonej pętli jest sprawdzenie rozmiaru zbioru stanów dla dwóch następujących po sobie warstw. Jeśli warstwa nowsza składa się z większej liczby stanów, algorytm uzyskuje nowe informacje o świecie i jest przygotowany do dalszego trawersowania po grafie planującym, jednakże gdy ów liczby są sobie równe dochodzi do swego rodzaju stagnacji- wszystkie możliwe ruchy nie wprowadzają nowych informacji o świecie. Mimo poczucia, iż rzeczywisty mechanizm mógłby poprawnie wykrywać nieskończone generowanie planu, okazuje się, iż porównanie następujących po sobie poziomów stanów, co do wartości, jest zdecydowanie niewystarczające.

Wniosek 2.1 *GRAPHPLAN zawsze kończy swoje działanie.*

a co za tym idzie

Wniosek 2.2 *Odpowiedzi GRAPHPLANU dla zadanych danych wejściowych przyjmują jedynie dwie formy:*

1. *plan, którego realizacja prowadzi do uzyskania wskazanych celów*



2. informacja o tym, iż nie istnieje plan spełniający oczekiwania użytkownika

Powoduje to, iż GRAPHPLAN jako algorytm jest kompletny i zawsze zakończy swoje działanie w skończonym czasie. Jest to własność, której większość z planerów korzystających z częściowego porządku nie posiada w swoim arsenale.

Rozdział 3

Programowanie ograniczeń

3.1 Wprowadzenie

Programowanie ograniczeń (inna nazwa: technologia więzów) jest narzędziem wykorzystywanym do rozwiązywania problemów z dziedzin kombinatoryki, sztucznej inteligencji, czy planowania jak i harmonogramowania zadań. W skład tego podejścia do programowania często wyróżnia się dwa elementy: ograniczenia (zwane również stałymi) oraz problem rozwiązywania ograniczeń (ang. Constraint Satisfaction Problem, CSP). Poniżej dokonano formalnego zdefiniowania powyższych komponentów:

Definicja 3.1 *Problem rozwiązywania ograniczeń, CSP jest następującą trójką:*

$$CSP = (V, D, C) \quad (3.1)$$

gdzie:

$V = \{x_1, x_2, \dots, x_n\}$ oznacza zbiór zmiennych wykorzystywanych do opisu problemu

$D = \{d_1, d_2, \dots, d_n\}$ oznacza zbiór dziedzin wyżej wspominanych zmiennych. W ramach rozważań zawartych w rzeczonej pracy rozpatrywane będą takie d_i , które są zbiorami zawierającymi skończoną liczbę potencjalnych wartości zmiennej x_i .

$C = \{c_1, c_2, \dots, c_m\}$ oznacza zbiór ograniczeń

Definicja 3.2 *Ograniczeniami (inne nazwy: stałe, więzy) nazywamy zmienne oraz zależności między nimi, które muszą zostać spełnione w ramach rozwiązywania problemu ograniczeń. Określamy je przy pomocy pary:*

$$C = (S, R) \quad (3.2)$$

gdzie:

S jest krotką wszystkich zmiennych wchodzących w skład relacji

R jest relacją, która definiuje jakie wartości mogą przyjmować zmienne, które w niej uczestniczą.

Relacje często przedstawia się przy pomocy zbioru zawierającego krotki, które składają się ze wszystkich przyporządkowań wartości do odpowiednich zmiennych. Przy następujących definicjach oczekiwany celem będzie utworzenie mechanizmu rozwiązywającego zadany problem. Jego wynikiem będzie zbiór wszystkich zmiennych, oraz krotek, które będą zawierały odpowiednie wartości przyporządkowane dla zmiennych.

Definicja 3.3 *Krotka (ang. tuple) - struktura danych, która w systemach informatycznych odzwierciedla uporządkowany ciąg wartości*

Dodatkowo wprowadza się termin **arność** ograniczenia:

Definicja 3.4 *Arność ograniczenia (ang. arity) związana jest z liczbą unikalnych zmiennych, która w nią wchodzi.*

Najpopularniejszymi typami ograniczeń są:

1. Ograniczenia o arności 1, zwane ograniczeniami **unarnymi** (w tym przypadku z reguły będą one ściśle związane z dziedziną, raczej nie będą występować w kontekście ograniczeń)
2. Ograniczenia o arności 2, zwane ograniczeniami **binarnymi**
3. Ograniczenia o arności 3, zwane ograniczeniami **ternarnymi**

Tak jak ograniczenie może mieć swoją arność, tak dla CSP również zdefiniowano pojęcie arności w lekko zmodyfikowany sposób

Definicja 3.5 *Arność CSP o wartości i zawiera w sobie wszystkie typy ograniczeń od arności 1 aż do arności i*

Wedle powyższego binarne CSP zawiera w sobie jedynie ograniczenia unarne jak i binarne.

Przykład 3.1 *Niech będzie dane równanie $x + y = z$, gdzie $x,y,z \in \{0,1\}$. Łatwo zauważyc, iż zadane równanie jest automatycznie ograniczeniem wpływającym na prezentowane zmienne. Przyporządkowując odpowiednie wartości do zbiorów z definicji otrzymano*

1. $V = \{x,y,z\}$
2. $D = \{d_x, d_y, d_z\}$, gdzie $d_x, d_y, d_z = \{0,1\}$
3. $C = \{x + y = z\}$

Arność ograniczenia występujące w przedstawionym przykładzie wynosi 3, determinowane jest to liczbą zmiennych, która wchodzi w jej skład. Rozwiązaniem tego problemu będą następujące dopasowania:

$$((x,y,z), \{0,0,0\}, \{1,0,1\}, \{0,1,1\})$$

Odnalezienie rozwiązań z przykładu 3.1 było trywialne ze względu na małą liczbę zmiennych, wąskie dziedziny oraz tylko jedno wprowadzone ograniczenie. Podobnie jak z planowaniem, wprowadzenie dodatkowych zmiennych, powiększanie dziedzin oraz zbioru ograniczeń znacznie wpływa na skomplikowanie odnajdowania rozwiązania.

3.2 Pojęcie ustalenia i spójności

W ostatecznej formie problem ograniczeń wyszukuje rozwiązanie przy pomocy ustalenia wartości zmiennych, jednakże naistotniejsza jest droga, jaką pokonuje, aby ów ustalenia uzyskać. W tej sekcji należy wprowadzić kilka dodatkowych definicji:

Definicja 3.6 *Ustalenie zmiennych jest spójne, gdy nie jest w konflikcie z żadnym z ograniczeń.*

Ustalenie spójne również w nomenklaturze programowania ograniczeń nazywane jest ustaleniem **legalnym**. Nadanie zmiennym X oraz Y wartość 1 w przykładzie 3.1 prowadzi do konfliktu, gdyż nie istnieje wartość 2 w zbiorze D_z .

Definicja 3.7 *Kompletnym ustaleniem nazywamy takie ustalenie, w którym wszystkie zmienne posiadają ustaloną wartość.*

Łatwo zauważyc, iż kompletne ustalenie jest jednocześnie rozwiązaniem problemu ograniczeń. Z tego płynie następujący wniosek:

Wniosek 3.1 *Każde rozwiązanie problemu ograniczeń jest spójne.*

Dodatkowo w źródłach [9] definiuje się częściowe ustalenie oraz częściowe rozwiązanie. Zgodnie z nazewnictwem częściowe ustalenie związane jest z sytuacją, gdy jeszcze nie wszystkie zmienne mają dokładnie określone wartości, natomiast częściowe rozwiązanie w praktyce identyfikuje się jako spójne częściowe ustalenie.

Po wprowadzeniu powyższych definicji należy rozpocząć rozważania na temat tego, w jaki sposób problem ograniczeń może zostać rozwiązany. Pierwszym z podejść może być ustalenie wartości dla zmiennych poprzez analizę ograniczeń, jakie między nimi występują. Ten proces nazywany jest **propagacją ograniczenia**.



Dzięki podstawowej analizie ograniczeń algorytm może wyeliminować wartości nadmiarowe znajdujące się w zadanych dziedzinach, co znacząco wpłynie na przyszłościowe osiągi pod kątem czasowym. Często propagacja ograniczeń jest wykonywana jako *preprocessing step*, czyli jako krok, który zostanie wykonany jeszcze przed rozpoczęciem prawdziwej pracy nad problemem.

Przy rozwiązywaniu problemów związanych z ograniczeniami bardzo intuicyjną reprezentacją, jest reprezentacja w formie *grafa*. Poprzez wierzchołki oznacza się zmienne wchodzące w skład problemu, natomiast poprzez krawędzie- binarne ograniczenia, która między nimi występują. Z tego powodu pożądany zabiegiem będzie przedstawienie wszystkich ograniczeń n-arnych w formie binarnej.

Kluczem do uzyskania poprawnego efektu propagacji ograniczeń jest skorzystanie z pojęcia zdefiniowanego jako **lokalna spójność**. Istnieją różne typy lokalnej spójności:

- **Spójność wierzchołkowa** (ang. node consistency) - Zmienna jest wierzchołkowo spójna, gdy wszystkie wartości znajdujące się w jej dziedzinie spełniają zdefiniowane przez nią ograniczenie unarne. Graf jest wierzchołkowo spójny, gdy wszystkie wierzchołki wchodzące w skład grafu są wierzchołkowo spójne. Zachowanie wierzchołkowej spójności z reguły sprowadza się do **zawężania** dziedziny zmiennej.
- **Spójność krawędziowa** (ang. arc consistency/edge consistency)- Zmienna jest spójna krawędziowo wtedy, gdy każda wartość z jej dziedziny spełnia binarne ograniczenia zmiennej. Graf jest krawędziowo spójny gdy każda para zmiennych jest ze sobą krawędziowo spójna.
- **Spójność ścieżki** (ang. path consistency) - Dwie zmienne a,b są spójne w kontekście ścieżki z trzecią zmienną c , gdy każde przypisanie wartości do zmiennych a,b spełniające ograniczenie występujące między rzecznymi zmiennymi dodatkowo spełnia ograniczenie między zmiennymi a,c oraz c,b .
- **K-spójność**- CSP jest k-spójne, gdy dla każdego spójnego ustalenia zawierającego $k - 1$ zmiennych można dołączyć k-tą zmienną bez załamania spójności. Pojęcie 2-spójność jest tożsame z spójnością krawędziową, a 3-spójność- ze spójnością ścieżki.

Sprowadzanie problemu ograniczeń do sytuacji, w której zachodzi lokalna spójność jest określane mianem **propagacją ograniczeń**. Jest to o tyle istotne, iż jedną z głównych metod rozwiązywania problemu ograniczeń jest sprowadzenie dziedzin zmiennych do pojedynczej wartości- wtedy całkowitym ustaleniem jest nadanie zmiennej jedynej wartości w swojej dziedzinie. Propagacja ograniczeń jest silnym mechanizmem wykrywającym niespójności- jeśli przy próbie propagacji któraś z wartości zostałaby z pustą dziedziną, wtedy należałyby przerwać rozwiązywanie problemu wraz ze zwróceniem informacji o fakcie, iż nie ma takiego ustalenia zmiennych przy obecnych dziedzinach, dla którego rozpatrywane ograniczenie jest rozwiązywalne.

3.3 Ograniczenia globalne

W teorii programowania ograniczeń istnieją również ograniczenia zwane **globalnymi**. Mimo swojej nazwy, globalne ograniczenia nie zawsze są związane ze wszystkimi zmiennymi wchodzącymi w skład problemu ograniczeń. Ich istnienie warunkowane jest występowaniem w świecie rzeczywistym zależności, które często się powtarzają i nie są unikalne dla jednego problemu. Przykładem takiego ograniczenia jest sytuacja, w której każda ze zmiennych ma mieć inną wartość. Formalnie mówiąc, wszystkie zmienne muszą być parami różne. Ów ograniczenie jest na tyle popularne, iż nosi ono swoją nazwę **Alldiff** i jest często ograniczeniem wbudowanym w moduły zajmujące się programowaniem ograniczeń, co ułatwia i przyspiesza pracę użytkownika. Innym ograniczeniem z rodziny ograniczeń globalnych jest **ograniczenie zasobów**. Jak sama nazwa wskazuje, rzeczone ograniczenie globalne wykorzystywane jest w modelowaniu sytuacji z dziedzin planowania bądź harmonogramowania zadań. Przy rozwiązywaniu tego typu ograniczenia częstą metodą jest sumowanie wartości zmiennych wchodzących w skład dziedziny [9].

3.4 Wyszukiwanie rozwiązań

Podstawowa metoda rozwiązywania problem z dziedziny ograniczeń nosi miano metody **cofającej** (ang. backtracking). Działa ona zbliżenie do mechanizmu przeszukiwania grafu wgłęb. Na początku wybierana jest jedna ze zmiennych. Dla każdej wartości z dziedziny dochodzi do częściowego ustalenia- zmienna otrzymuje



wartość równą pierwszej wartości w swojej dziedzinie. Następnie, korzystając z tej informacji dochodzi do propagacji ograniczeń. Jeśli wybrana została poprawna wartość propagacja ograniczeń doprowadzi do znalezienia rozwiązania problemu. Istnieje również sytuacja, w której wybrana wartość prowadzi do slepego zaułka, czyli do sytuacji, w której nie istnieje odpowiednie ustalenie zmiennych. Wtedy należy **cofnąć** się do miejsca, w którym zmiennej nadaliśmy wartość i spróbować innego ustalenia. Połączenie mechanizmu cofnięcia wraz z propagacją ograniczeń, czyli z utrzymywaniem lokalnej spójności na każdym etapie wyszukiwania rozwiązania jest znakomitą techniką poprawiającą wydajność. Dzięki propagacji ograniczeń często dochodzi do sytuacji, w której wyżej wymienione ślepe zaułki są eliminowane zanim mechanizm cofania je rozpatrzy.

Drugą metodą jest metoda nazywana **wyszukiwaniem lokalnym** (ang. local search). Metoda cofająca, jak sama nazwa wskazuje, próbowała dokonać częściowego ustalenia, by następnie przy propagacji ograniczeń udowodnić, iż rozpatrywane częściowe ustalenie jest prawidłowe i generuje odpowiednie całkowite ustalenie. Metodologia wyszukiwania lokalnego różni się w swojej filozofii tym, iż na samym początku dochodzi do pełnego ustalenia zmiennych. W znacznej większości przypadków ów pełne ustalenie jest nieprawidłowe, to znaczy nie spełnia wszystkich ograniczeń. Wtedy mechanizm próbuje na bieżąco naprawiać sytuację modyfikując całkowite ustalenie w takich sposób, aby ustalenie nadal było całkowite jednocześnie spełniając ograniczenie, które wcześniej powodowało konflikty. Jeśli algorytm będzie w stanie rozwiązać wszystkie ograniczenia uzyska odpowiednie ustalenie zmiennych.

Porównując powyższe dwie metody łatwo zauważyc, iż różnią się one od siebie w znacznym stopniu, nie tylko w samej filozofii działania, lecz także w efektach.

Definicja 3.8 *Algorytmem kompletnym jest algorytm, który gwarantuje uzyskanie rozwiązania oraz jest w stanie wykryć, gdy takowe rozwiązanie nie istnieje*

Przeciwieństwem algorytmu kompletnego jest algorytm **niekompletny**, czyli taki, który nie gwarantuje uzyskania optymalnego rozwiązania oraz wykrycia, czy problem posiada rozwiązanie. Algorytmy niekompletne natomiast są o wiele szybsze oraz dobrze przybliżają optymalne rozwiązanie.

Zgodnie z powyższym metoda cofająca jest przykładem algorytmu kompletnego- systematycznie generuje nowe ustalenia oraz propagacje ograniczeń, natomiast wyszukiwanie lokalne jest przykładem algorytmu niekompletnego- łatwo sobie wyobrazić sytuację, iż naprawienie jednego ograniczenia może nieustannie generować zepsucie kolejnego. W trakcie wyboru algorytmu ważnym jest, aby znać jego zalety jak i wady. Do implementacji GRAPHPLANU został wykorzystany mechanizm cofania, aby zwracany przez algorytm plan był zawsze optymalny.

Powyżej wymienione metody posiadają wiele dodatkowych usprawnień oraz specjalnie zdefiniowanych heurystyk, z którymi czytelnik może zapoznać się kierując się do następującej pozycji w bibliografii [6]

3.5 Programowanie w logice z ograniczeniami

Ze względu na wiele podobieństw w mechanizmach, jak i w samej idei, między technologią więzów a programowanie w logice, takich jak chociażby mechanizm cofania, który jest powszechnie wykorzystywany w obu podejściach, zdecydowano się na utworzenie połączenia między nimi w celu uzyskania korzystniejszych wyników. Z tej kombinacji powstał typ programowania nazywany **Programowaniem w logice z ograniczeniami**.

Definicja 3.9 *Programowanie w logice z ograniczeniami* (ang. Constraint logic programming) jest jedną z form programowania ograniczeń, w której podstawowe mechanizmy programowania w logice zostały rozszerzone o konstrukcje pochodzące z programowania ograniczeń.

Przykład 3.2 Programowanie ograniczeń zastosowane w języku PROLOG, który jest językiem programowania logicznego:

```
FUNC(X,Y) :-  
    X+Y > 0,  
    writeln(X),  
    writeln(Y).
```

Kod źródłowy 1: Metoda wypisująca liczby gdy ich suma jest większa od 0



W powyższym przykładzie zastosowano najprostsze z możliwych ograniczeń: wartość sumy zmiennych musi być większa od 0. Ze względu na częste parowanie programowania ograniczeń z programowaniem w logice, wiele implementacji języków programowania w logice implementuje gotowe moduły, które dostarczają technologię więzów dla różnych typów dziedzin. Dla przykładu, dla jednej z najpopularniejszych implementacji języka programowania PROLOG dostępne są następujące moduły:

- clpf - moduł, w którym dziedziny są skończone i składają się z liczb całkowitych. Ów moduł wykorzystywany jest w zaimplementowanym algorytmie GRAPHPLAN
- clpb - moduł, w którym dziedziny składają się z wartości boolowskich
- clpq - moduł, w którym dziedziny składają się z liczb wymiernych w formie dziesiętnej
- clpr - moduł, w którym dziedziny składają się z liczb rzeczywistych zmiennopozycyjnych

Dzięki powyższym modułom można tworzyć bardziej zmyślne ograniczenia. Z przykładami takich zastosowań czytelnik będzie mógł się zapoznać w sekcji opisującej implementację algorytmu.

3.6 Obrazowe przykłady

3.6.1 Problem plecakowy

Przykładem realizacji problemu ograniczeń poprzez wykonanie operacji propagacji ograniczeń wraz z mechanizmem cofania będzie popularny **Problem plecakowy**

Przykład 3.3 Dyskretny problem plecakowy- (ang. *discrete knapsack problem*) problem wyboru przedmiotów w taki sposób, aby spełniały następujące założenia:

- suma wartości wybranych przedmiotów musi być jak największa
- suma wag wybranych przedmiotów nie może przekraczać wartości plecaka

oraz został osiągnięty następujący cel: suma wartości wybranych przedmiotów ma być jak największa.

Pierwszym podejściem, automatycznie nasuwającym się przy próbie rozwiązania tego problemu, jest iteracyjne pakowanie przedmiotów o najwyższej wartości aż do zajścia sytuacji, w której plecak nie jest w stanie pomieścić następnego przedmiotu tego typu. Wtedy dochodzi do zmiany przedmiotu na ten, który jest drugi co do wartości oraz powtórzenia dla niego wyżej opisanego procesu. Łatwo zauważać, iż takie rozwiązanie, mimo tego, że zawsze wyprodukuję jakiś wynik, jest mało optymalne. Mogą zdarzyć się sytuacje, w których przedmiot o dużej wartości jest na tyle ciężki, że warto byłoby z niego zrezygnować na rzecz mniej wartych, ale za to lżejszych przedmiotów. Takie sytuacje nie zostaną wychwycone przez takowe podejście. Przedstawiony powyżej sposób rozwiązania problemu nosi nazwę rozwiązania **zachłannego**, które cechuje się wyszukiwaniem lokalnego, a nie globalnego, **ekstremum** funkcji.

Łatwo zauważać, iż sposób zachłanny jest modelowany na wzór zachowania człowieka postawionego we wskazanej sytuacji. Drugie podejście natomiast wykorzystuje analityczne podejście, które w świecie informatycznym nosi nazwę **programowania dynamicznego**. Ten typ programowania opiera się na podziale występującego problemu na mniejsze podproblemy względem ustalonych parametrów. Dla wskazanego problemu definiuje się ciąg wag jako w_1, \dots, w_n oraz wartości v_1, \dots, v_n , gdzie n oznacza liczbę branych pod uwagę przedmiotów, oraz funkcję $A(i,j)$, gdzie i oznacza liczbę rozpatrzonych elementów, a j - maksymalną wartość, którą można otrzymać z wskazanych elementów. Problem srowadza się do wyznaczenia wartości $A(n,W)$, gdzie W oznacza maksymalną wagę plecaka. Funkcja $A(i,j)$ definiowana jest następującym wzorem:

$$\begin{aligned} A(0,j) &= 0 \\ A(i,0) &= 0 \\ A(i,j) &= A(i-1,j) \text{ dla } w_i > j \\ A(i,j) &= \max(A(i-1,j), A(i-1,j-w_i) + v_i) \text{ dla } w_i \leq j \end{aligned} \tag{3.3}$$

Funkcja ta nosi nazwę *funkcji rekurencyjnej Bellmana*

Alternatywnym metoda do skomplikowanego podejścia w programowaniu dynamicznym jest użycie wcześniej wspomnianego programowania ograniczeń. Ograniczenie na liczbę przedmiotów prezentowane jest w postaci $w_1 * A + w_2 * B + \dots = < W$, gdzie współczynniki A, B, \dots to liczba elementów do wzięcia. Następnie należy zmaksymalizować etykietowanie współczynników A, B, \dots względem funkcji $v_1 * A + v_2 * B + \dots$.

3.6.2 Krypto-arytmetyczna łamigłówka

Krypto-arytmetyczna łamigłówka jest przykładem matematycznego równania, w którym cyfry zastąpiono przez litery. Zadaniem rozwiązującego jest zamiana odpowiednich liter na cyfry w taki sposób, aby prezentowane równanie było matematycznie poprawne.

$$\begin{array}{r} TWO \\ + TWO \\ \hline FOUR \end{array}$$

Rysunek 3.1: Przykładowa łamigłówka krypto-arytmetyczna

Przykład 3.4

Najważniejszymi założeniami łamigłówki są:

- Wartości ukryte pod literami T,W,O,F,U,R są parami różne
- Najbardziej znacząca cyfra jest niezerowa (w tym przypadku $T \neq 0$ oraz $F \neq 0$)

Pierwszy sposób na rozwiązanie, bez użycia programowania ograniczeń:

```
digit(0). digit(1). digit(2). digit(3). digit(4).
digit(5). digit(6). digit(7). digit(8). digit(9).

riddle([T,W,O],[T,W,O],[F,O,U,R]) :-
    digit(T), digit(W), digit(O),
    digit(F), digit(U), digit(R),
    T =\= W, T =\= O, T =\= F, T =\= U, T =\= R,
    W =\= T, W =\= O, W =\= F, W =\= U, W =\= R,
    O =\= T, O =\= W, O =\= F, O =\= U, O =\= R,
    F =\= T, F =\= W, F =\= O, F =\= U, F =\= R,
    U =\= T, U =\= W, U =\= O, U =\= F, U =\= R,
    R =\= T, R =\= W, R =\= O, R =\= F, R =\= U,
    T > 0, F > 0,
    100*T + 10*W + O + 100*T + 10*W + O =:= 1000*F + 100*O + 10*U + R.
```

Kod źródłowy 2: Implementacja rozwiązania łamigłówki krypto-arytmetycznej bez użycia programowania ograniczeń

Drugi sposób na rozwiązanie problemu, z użyciem programowania ograniczeń:

```
: - use_module(library(clpf)).  
  
riddle([T,W,O],[T,W,O],[F,O,U,R]) :-  
    Values = [T,W,O,F,U,R],  
    Values ins 0..9,  
    all_different(Values),  
    T #> 0,  
    F #> 0,  
    100*T + 10*W + O + 100*T + 10*W + O #= 1000*F + 100*O + 10*U + R,  
    label(Values).
```

Kod źródłowy 3: Implementacja rozwiązywania łamigłówki krypto-arytmetycznej z użyciem programowania ograniczeń

Otrzymane przykładowe rozwiązanie $T = 7, W = 3, O = 4, F = 1, U = 6, R = 8$.

$$\begin{array}{r} 7 \ 3 \ 4 \\ + \ 7 \ 3 \ 4 \\ \hline 1 \ 4 \ 6 \ 8 \end{array}$$

Rysunek 3.2: Przykładowe rozwiązanie wskazanej łamigłówki

Porównanie osiągnięć wskazanych rozwiązań:

Sposób	Wnioskowania	Czas[s]
Bez CLP	8 679 856	0.449
Z CLP	19 119	0.002

Tablica 3.1: Otrzymane wyniki dla znalezienia pierwszego rozwiązania łamigłówki

3.7 Wykorzystanie w algorytmie

Podczas opisywania mechanizmów stojących za programowaniem ograniczeń często wykorzystywana strukturą był graf, chociażby w sekcji omawiającej pojęcie lokalnych spójności (3.2). Ze względu na ów powiązanie między programowaniem ograniczeń a GRAPHPLAN'em do podstawowego opisu GRAPHPLANU z rozdziału 2.2 dodano funkcjonalności opisane w powyższych rozdziałach.

Każdy ze stanów oraz akcji zawiera w sobie dodatkowy **indykator**. Jest to liczba ze zbioru liczb całkowitych, o której należy myśleć bardziej w kontekście wartości bool'owskich *{prawda,fałsz}*. Wartość liczby równa 0 indukuje fałszywość stanu, natomiast wartość większa od 0 indukuje jego prawdziwość. Przy pomocy indykatorów program ustala, które stany, bądź akcje są w danej warstwie prawdziwe, czyli występują w świecie oraz takie, które w ów świecie w danym momencie nie występują, czyli są fałszywe. Odbywa się to w następujący sposób:

1. Wszystkie stany wchodzące w stan początkowy otrzymują indykator równy 1, gdyż są aktualnie prawdziwe w rozpatrywanym świecie.
2. W trakcie generowania akcji następuje utworzenie powiązania między warunkami zajścia, akcjami oraz ich efektami. Akcja zostaje powiązana ze swoim warunkiem następującym ograniczeniem: wartość indykatora akcji jest mniejsza bądź równa wartości indykatora warunku. Należy to rozumieć w następujący sposób: jeśli warunek jest prawdziwy to akcja **może** zachodzić w świecie, natomiast jeśli warunek jest nieprawdziwy, czyli ma indykator równy 0, akcja automatycznie dostaje indykator równy 0. Efekt zostaje powiązany ze swoją akcją poprzez następujące ograniczenie: jeśli jakakolwiek akcja w danej



warstwie, która ma dany stan za efekt zachodzi, wtedy również i efekt w nim występuje. Jeśli wszystkie akcje generujące ów efekt mają indykatorem równy 0 wtedy efekt nie może zachodzić na danym poziomie, więc również otrzymuje indykatorem równy 0.

3. Dochodzi do sprawdzenia relacji wzajemnego wykluczania poprzez sprawdzenie indykatatorów dwóch stanów- Jeśli ich iloczyn jest równy 0, wtedy dwa stany nie mogą razem występować na danym poziomie.
4. Przy dokładnej realizacji kroków 2 i 3 algorytm jest w stanie wygenerować kolejny poziom stanów. Po wygenerowaniu dochodzi do sprawdzenia, czy wszystkie stany zawarte w zbiorze przechowującym cele mają indykatory równe 1. Jeśli nie, rzeczony proces jest powtarzany aż do otrzymania pożądanego skutku.
5. Gdy wszystkie cele otrzymają indykatorem 1, program przelicza wszystkie indykatory, dzięki czemu jest w stanie bezbłędnie określić, który stan bądź akcja na danym etapie przetwarzania świata znajdują się w nim bądź nie. Ów mechanizm jest silnie wykorzystywany przy generowaniu grafów, przedstawiających zachodzące w świecie zmiany.

Dzięki tej z pozoru niewielkiej modyfikacji algorytmu GRAPHPLAN zyskuje on zdecydowane przyśpieszenie w fazie kreowania planu. Gdy GRAPHPLAN dojdzie do odpowiedniego poziomu stanów, gdzie znajdują się wszystkie stany ze zbioru celów, wszystkie komponenty wchodzące w skład grafu planującego będą miały dodatkową informację o swojej prawdziwości w danej warstwie. Odnalezienie planu sprowadza się do rozwiązania problemu ograniczeń dla wszystkich stanów grafu. Dokonywane jest to wedle myśli przewodniej GRAPHPLANU, czyli poprzez mechanizm cofania wraz z propagacją ograniczenia przy zachowaniu lokalnej spójności. Bez tej modyfikacji wyłuskanie planu z grafu planującego przypominałoby wyszukiwanie w głąb w grafie. Przechowywanie dodatkowej informacji w formie indykatora znacznie usprawnia ten proces.

Dokładne sformuowanie ograniczeń przy pomocy symboli matematycznych odbędzie się w sekcji poświęconej implementacji algorytmu.

Rozdział 4

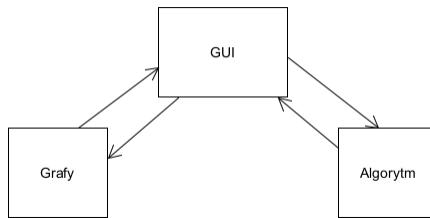
Implementacja

4.1 Opis komponentów i ich połączeń

W programie implementującym algorytm będący obiektem badań można wyróżnić 3 główne komponenty:

- **Algorytm** zaimplementowany w języku programowania **PROLOG**
- **Generator grafów** zaimplementowany w formie modułu w języku programowania **Python**
- **Warstwa graficzna (GUI)** zaimplementowana w języku programowania **Python** przy użyciu biblioteki **Tkinter**

Naistostniejszą z perspektywy użytkownika jest **Warstwa graficzna**. Jest to program, którego uruchomienie pozwala na interakcję z utworzonym narzędziem w przyjazny dla użytkownika sposób. Istnieje również możliwość uruchomienia algorytmu z pominięciem warstwy graficznej, o czym można dowiedzieć się więcej w sekcji 4.5. Z wyświetlonego menu użytkownik może wybrać przykładowe, wcześniej spreparowane światy, zdefiniować swój stan początkowy oraz cel (w niektórych światach cel jest intuicyjny, więc jego definicja nie jest od użytkownika wymagana) i uruchomić algorytm. Wynikiem działania programu jest plan, który wyświetlony jest w formie tekstuowej z opisem na kroki oraz dwa grafy: graf pełen (zawierający wszystkie składowe opisane w 2) oraz graf uproszczony, który zawiera jedynie niezbędne stany oraz akcje wymagane do zrozumienia wygenerowanego planu. Poniższy schemat klarownie przedstawia relacje między komponentami w trakcie działania programu:

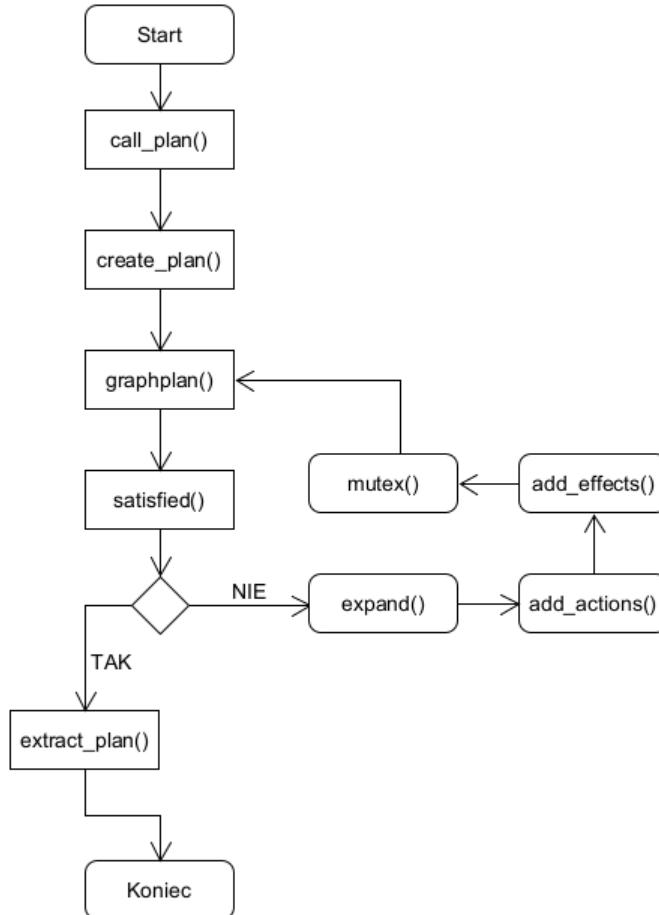


Rysunek 4.1: Zależności między komponentami

Główna jednostką sterującą jest warstwa graficzna. W momencie naciśnięcia przez użytkownika odpowiedniego przycisku generującego rozwiązanie, warstwa graficzna zbiera wszystkie niezbędne informacje: w jakim świecie użytkownik pracuje, w jaki sposób zdefiniował warunki początkowe, oraz jakie cele chce on uzyskać. Następnie obrobione informacje przesypane są do algorytmu, który poza wygenerowaniem planu, do pliku tekowego wypisuje stany, akcje oraz mutexy dla każdej z warstw. Następnie algorytm przesyła swoją odpowiedź do GUI, który wysyła żadanie wygenerowania grafów do odpowiednich komponentów odpowiedzialnych za ich generowanie. W skład komponentu "Grafy" wchodzą dwie klasy, przy czym każda z nich generuje unikalny graf.

4.2 Implementacja algorytmu

Algorytm **GRAPHPLAN** został zaimplementowany w języku programowania PROLOG. Ponadto wykorzystuje wbudowany moduł **clpfd** do implementacji programowania ograniczeń dla Poniższe diagramy przedstawiają proces generowania pojedynczego planu.



Rysunek 4.2: Główna pętla algorytmu

Definicja 4.1 *Klauzula – przedstawiona informacja o świecie*

Definicja 4.2 *Fakt – informacja o świecie, która jest zawsze prawdziwa*

Definicja 4.3 *Reguła – informacje, które są prawdziwe, po spełnieniu pewnych warunków*

Definicja 4.4 *Predykat – sposób wyrażania warunków w programowaniu w logice*

Przed rozpoczęciem omawiania struktury programu wprowadzono podstawowe definicje stosowane w programowaniu w logice. Przykładem klauzuli jest zdanie *kobieta(anna)*. informujące program o tym, iż Anna jest kobietą. Ze względu na sposób reprezentacji danych w języku programowania PROLOG, wszystkie stałe określane są małymi literami, gdyż ciągi znaków zaczynające się wielką literą zarezerwowane są dla *zmiennych*. Należy również zwrócić uwagę na wykorzystanie znaku przystankowego w postaci kropki (.), przedstawiający koniec przekazywanej informacji. *kobieta(anna)*. również jest przykładem jednoargumentowego predykatu. Predykaty jednoargumentowe często określany są mianem *własności*. Predykaty o większej liczbie argumentów często nazywane są *relacjami*. W Prologowej nomenklaturze predykaty zapisują się zgodnie z następującą

strukturą: X/Y , gdzie X oznacza nazwę relacji (nagłówek), a Y liczbę argumentów przyjmowaną przez opisywany predykat. Zgodnie z powyższym zapis *kobieta/1* oznacza, iż relacja kobieta przyjmuje jeden argument. Zdanie *kolor(brzowy)*. jest przykładem faktu, czyli zdania zawsze prawdziwego. Fakty różnią się od reguł tym, iż reguły nie muszą być zawsze prawdziwe [4]. W sekcji 2.4 wprowadzono przykład reguły, wraz z opisem jej struktury.

Założeniem działania algorytmu jest poprawne zdefiniowanie warunków początkowych oraz celów, przy użyciu wcześniej spreparowanych przez użytkownika relacji. Ponadto zbiór akcji również musi zostać bezpośrednio utworzony przez użytkownika, algorytm w trakcie działania nie dokonuje żadnego sprawdzenia zgodności wprowadzonych danych. Z tego powodu warstwa graficzna programu udostępnia tylko część środowisk, które zostały opisane przez autora pracy zgodnie z wytycznymi języka opisu STRIPS.

Predykat o nazwie *call_plan/2* odpowiada za utworzenie pliku tekstowego (nazwa pliku: *output.txt*) oraz przekierowanie strumienia danych do ustalonego pliku. Następnie pobiera stan świata, które zdefiniowany jest w formie faktu *initial_state/1*, który przyjmuje jeden argument w formie listy zawierającej wszystkie stany początkowe. Następnie dochodzi do uruchomienia predykatu *create_plan/3*, który przyjmuje jako dane wejściowe początkowy stan oraz wymagany cel, a wynikiem jego działania jest utworzony plan. W tym predykatie, z pomocą predykatu *findall/3*, odbywa się nadanie wszystkim stanom początkowym indykatora jeden, gdyż wszystkie z tych stanów są prawdziwe w przedstawionym świecie. Następnie dochodzi do wykreowania pierwszego zbioru akcji przy pomocy wbudowanego predykatu *setof/3*. Po wykreowaniu odpowiednich zbiorów dochodzi do uruchomienia predykatu *graphplan/4* - głównej pętli programu.

```
% call_plan(+Goals,-Plan)
call_plan(Goals,Plan) :-
    tell('outputs/output.txt'),
    initial_state(S),
    create_plan(S,Goals,Plan),
    write("Plan: "), writeln(Plan),
    told.
```

Kod źródłowy 4: Implementacja predykatu *call_plan/2*

```
% create_plan(+StartState,+Goals,-Plan)
create_plan(StartState, Goals, Plan) :-
    findall(State/1, member(State,StartState), StartLevel),
    setof(action(Action, Precondition, Effects),
        (effects(Action,Effects), preconditions(Action,Precondition)),
        AllActions),
    write("StartLevel: "), write(StartLevel), nl,
    graphplan([StartLevel], Goals, Plan, AllActions).
```

Kod źródłowy 5: Implementacja predykatu *create_plan/2*

Działanie predykatu *graphplan/4* można podzielić na dwa etapy. W pierwszym etapie sprawdzany jest warunek konieczny utworzenia grafu- obecność wszystkich stanów ze zbioru celów na aktualnym poziomie stanów. Dokonywane jest to przy pomocy predykatu *satisfied/2*, który iteruje po każdym celu z listy celów i dokonuje sprawdzenia wskazanego warunku. Dodatkowo rzeczona klauzula ustala, czy indykatorka każdego z celów jest większy od zera, czyli czy jest możliwość jego wykonania na danym poziomie stanów. Jeśli powyższe warunki są spełnione dla każdego celu, algorytm przechodzi do predykatu *extract_plan/2*, który na podstawie aktualnego poziomu stanów podejmuje próbę generowania odpowiedniego planu.

```
% graphplan(+GraphPlan, +Goals, +AllActions, -Plan)
graphplan([StateLevel | GraphPlan], Goals, AllActions, Plan) :-
    satisfied(StateLevel, Goals),
    extract_plan([StateLevel | GraphPlan], Plan)
;
expand(StateLevel, ActionLevel, NewStateLevel, AllActions),
graphplan([NewStateLevel, ActionLevel, StateLevel | GraphPlan],
Goals, AllActions, Plan).
```

Kod źródłowy 6: Implementacja predykatu graphplan/4

```
% satisfied(+StateLevel, +Goals)
satisfied(_, []).

satisfied(StateLevel, [G | Goals]) :-
    member(G/IG, StateLevel),
    IG #> 0,
    satisfied(StateLevel, Goals).
```

Kod źródłowy 7: Kod źródłowy implementacji predykatu satisfied/2

```
% extract_plan(+Graph, -Plan)
extract_plan([], []).

extract_plan([ChosenStates, ActionLevel | RestOfGraph], Plan) :-
    collect_vars(ActionLevel, AVars),
    labeling([], AVars),
    findall(A, (member(A/1, ActionLevel)), ChosenActions),
    extract_plan(RestOfGraph, RestOfPlan),
    write("ChosenActions: "), write(ChosenActions), nl,
    write("ChosenStates: "), writeln(ChosenStates),
    append(RestOfPlan, [ChosenActions], Plan).
```

Kod źródłowy 8: Implementacja predykatu extract_plan/2

W przeciwnym wypadku algorytm przechodzi do drugiej części, w której dochodzi do rozszerzenia grafu planującego o kolejne warstwy przy pomocy predykatu *expand/4*. Predykat *expand/4* ma za zadanie rozszerzyć graf o kolejną warstwę. Wykonuje to korzystając z dwóch predykatów, których nazewnictwo w pełni odpowiada funkcjonalności, jakie implementują. Są to- *add_action/6*, którego zadaniem jest generowanie oraz dodawanie akcji do poziomu akcji danej warstwy oraz *mutex_state/1* oraz *mutex_action/2*, odpowiedzialne za tworzenie relacji wzajemnie wykluczających między stanami jak i akcjami. W ramach predykatu *add_action/6* realizowany jest również predykat *add_effects/4*, który dla każdej akcji dodaje jej efekty do przyszłego poziomu stanów.



```
% expand(+StateLevel, -ActionLevel, -NextStateLevel, +AllActions)
expand(StateLevel, ActionLevel, NextStateLevel, AllActions) :-
    add_actions(StateLevel, AllActions, [], NewActionLevel, [], NewNextState),
    findall(action(zostan(P), [P], [P]), member(P, StateLevel), PersistActs),
    add_actions(StateLevel, PersistActs, NewActionLevel,
    ActionLevel, NewNextState, NextStateLevel),
    mutex_action(ActionLevel, NextStateLevel),
    mutex_list(NextStateLevel),
    write("ActionLevel: "), write(ActionLevel), nl,
    write("StateLevel: "), write(NextStateLevel), nl.
```

Kod źródłowy 9: Implementacja predykatu expand/4

Predykat *write/1* wykorzystywany jest wyłącznie w celu umieszczenia odpowiednich informacji o poziomie stanów oraz o poziomie akcji w pliku tekstowym, który następnie wykorzystywany jest podczas generowania odpowiednich grafów.

```
%add_actions(+StateLevel,+Action,+PreviousActionLevel,
-NewActionLevel,+PreviousStateLevel,-NewStateLevel)
add_actions(_,[],ActionLevel, ActionLevel, NextStateLevel, NextStateLevel).

add_actions(StateLevel, [action(A,Precondition,Effects) | Acts],
ActLev0, ActLev, NextLev0, NextLev) :-
    IA in 0..1,
    includes(StateLevel, Precondition, IA),
    add_effects(IA, Effects, NextLev0, NextLev1), !,
    write("A: "), write(A), nl,
    write("Precondition: "), write(Precondition), nl,
    write("Effects: "), write(Effects), nl,
    add_actions(StateLevel, Acts, [A/IA | ActLev0], ActLev, NextLev1, NextLev)
;
    add_actions(StateLevel, Acts, ActLev0, ActLev, NextLev0, NextLev).
```

Kod źródłowy 10: Implementacja predykatu add_actions/6

Przy implementacji predykatu *add.actions/6* należy zwrócić uwagę na wykorzystanie programowania ograniczeń w formie zmiennej **IA**, która przyjmuje wartość ze zbioru {0,1}, gdyż akcja może, ale nie musi być realizowana na danym poziomie akcji. Dodatkowo rzecznego predykat korzysta z predykatu *includes/3*, który dodaje warunek na wartość przyjmowaną przez identyfikator akcji- nie może być on większy niż identyfikator warunków zajścia. Należy to interpretować w następujący sposób: jeśli którykolwiek z warunków początkowych miałby w danym poziomie wartość równą 0 (czyli byłby nieprawdziwy), wtedy dana akcja nie może znajdować się w planie na wskazanym poziomie, natomiast prawdziwość wszystkich warunków (czyli identyfikatora równego 1 dla każdego warunku), powoduje iż akcja znajdzie się w zbiorze potencjalnych akcji do wykonania na wskazanym poziomie.

```
% includes(+StateLevel, +Preconditions, +Indykator)
includes(_,[],_).

includes(StateLevel, [P|Ps], IA) :-
    member(P/I, StateLevel),
    IA #=< I,
    includes(StateLevel, Ps, IA).
```

```
%add_effects(+Indykator,+PreviousStateLevel,-NextStateLevel)
add_effects(_,[],StateLevel,StateLevel).

add_effects(IA, [P | Ps], StateLev0, ExpandedState) :-
    (remove(P/IP, StateLev0, StateLev1), !,
     NewIP #= IP+IA,
     StateLevel = [P/NewIP | StateLev1]
    ;
     StateLevel = [P/IA | StateLev0], !
    ),
    add_effects(IA, Ps, StateLevel, ExpandedState).
```

Kod źródłowy 11: Implementacja predykatu add_effects/4

Następnie, w celu zachowania spójności, dla utworzonych zbiorów generowane są relacje wzajemnego wykluczania określone mianem *mutex’ów*. Mechanizm mutex’ów implementowany jest przy pomocy dwóch predykatów:

- **mutex_state/1** - predykat odpowiedzialny za ustalenie, które stany znajdują się w relacji wykluczającej na wskazanym poziomie generowania planu
- **mutex_action/2** - predykat odpowiedzialny za ustalenie, które akcje znajdują się w relacji wykluczającej na wskazanym poziomie generowania planu. Dodatkową funkcjonalnością rzeczone predykatu jest ustalenie mutex’ów między efektami akcji, które nie zawsze muszą być objęte definicjami wprowadzonymi w ramach **mutex_state/1**.

Działanie predykatów jest bardzo zbliżone z tą różnicą, iż operują na innych obiektach. Schemat działania prezentuje się w sposób następujący:

1. Ustal element ze zbioru stanów
2. Następnie dobierz do pary każdy z pozostałych elementów poziomu stanów i sprawdź, czy ów stany wykluczają się wzajemnie
3. Wróć do pierwszego kroku, jako element ustalając następny obiekt wchodzący w skład listy stanów

Powyzszy opis został zaprezentowani z perspektywy predykatu **mutex_state/1**. Dla predykatu **mutex_action/2** jedyną różnicą jest dodatnie ostatniego kroku, w którym wszystkie efekty wskazanych akcji zostają ze sobą połączone relacją wzajemnie wykluczającą. Determinowane jest to uniknięciem sytuacji określonej w sekcji 2.6.1 jako **niespójny efekt**.

```
%mutex_state(StateLevel)
mutex_state([]).

mutex_state([P | Ps]) :-
    mutex_single(P,Ps),
    mutex_state(Ps).
```

Kod źródłowy 12: Implementacja predykatu mutex_state/1

```
%mutex_single(+State,+StateLevel)
mutex_single(_,[]).

mutex_single(P/I, [P1/I1 | Rest]) :-
    ( mutex(P,P1), !, I*I1 #= 0
    ;
    true
    ),
    mutex_single(P/I,Rest).
```

Kod źródłowy 13: Implementacja predykatu mutex_single/2

Przy implementacji predykatu **mutex_single/2** należy zwrócić uwagę na moment, w którym dwa stany zostały uznane przez algorytm jako wykluczające. Wtedy dochodzi do ustalenia ograniczenia na indykatory tych zmiennych w postaci $I * I1 \#= 0$, co znaczy, iż wspomniane dwa stany nie mogą jednocześnie występować w danej iteracji świata (jeśli istniałyby, to każdy z tych stanów miałby indykator równy 1, a $1 * 1 = 1$, co jest sprzeczne z ustalonym ograniczeniem)

```
%mutex(+State1,+State2)
mutex(P,~P) :-
    write("Mutex: ["), write(P), write(",") ,write(~P), writeln("]"),!.

mutex(~P,P) :-
    write("Mutex: ["), write(~P), write(",") ,write(P), writeln("]"),!.

mutex(A,B) :-
    inconsistent(A,B),
    write("Mutex: ["), write(A), write(",") ,write(B), writeln("]"),
    !.
```

Kod źródłowy 14: Implementacja predykatu mutex/2

Warunki wykluczania stanów wynikają wprost z informacji zawartych w sekcji 2.6.1. Predykat *incosistent/2* jest bezpośrednio definiowany przez użytkownika w ramach ustalania warunków zachodzących w rozpatrywanym świecie.

W przypadku generowania wykluczeń dla akcji należy jedynie wspomnieć o predykatach: *mutex_for_action/3*, *mutex_all_states/3* oraz *apply_mutex/4*. Pozostałe predykaty, mimo posiadania innych nazw, działają identycznie względem predykatów utworzonych na rzecz ustalania relacji wykluczającej dla pary stanów.

```
%mutex_for_action(+Action1,+Action2,+StateLevel)
mutex_for_action(A1,A2,StateLevel) :-
    (preconditions(A1,Precondition), effects(A2,Effects)
    ;
    preconditions(A2,Precondition), effects(A1,Effects)
    ),
    member(P1,Precondition),
    member(P2,Effects),
    mutex(P1,P2),
    write("Mutex: ["), write(A1), write(",") ,write(A2), writeln("]"),
    mutex_all_states(A1,A2,StateLevel),
    !.
```

Kod źródłowy 15: Implementacja predykatu mutex_for_action/3



Predykat **mutex_for_action/3** sprawdza, czy dwie akcje sobie nie przeszkadzają (przeszkadzanie jest jednym z przypadków, który determinuje wykluczenie dwóch akcji. Więcej w sekcji 2.6.1) przez sprawdzenie, czy warunki zajścia jednej z akcji nie kolidują z efektami drugiej.

```
%mutex_all_state(+Action1,+Action2,+StateLevel)
mutex_all_states(A1,A2,StateLevel) :- 
    effects(A1,E1),
    effects(A2,E2),
    findall([X,Y],(member(X,E1),member(Y,E2)),C),
    apply_mutex(C,A1,A2,StateLevel).
```

Kod źródłowy 16: Implementacja predykatu mutex_all_states/3

Jeśli dwie akcje są ze sobą w relacji wykluczającej to ich efekty także muszą się w niej znaleźć. Wykonywane jest to przy pomocy powyższego predykatu **mutex_all_states/3**, który zbiera efekty dwóch akcji, tworzy z nich iloczyn kartezjański przy pomocy predykatów *findall/3* oraz *member/2*. Wygenerowany iloczyn przekazuje do predykatu **apply_mutex/4**, którego zadaniem jest wygenerowanie odpowiednich ograniczeń względem wskazanego zbioru efektów.

```
%apply_mutex(+EffectsSet, +A1,+A2,+StateLevel)
apply_mutex([],_,_,_).

apply_mutex([H | T], A1/I1,A2/I2,StateLevel) :-
    [First,Second] = H,
    member(First/F,StateLevel),
    member(Second/S,StateLevel),
    F*S #=< I1*I2,
    write("Mutex: ["), write(First), write(","), write(Second), writeln("]"),
    apply_mutex(T,StateLevel).
!.
```

Kod źródłowy 17: Implementacja predykatu apply_mutex/4

Należy zauważyć, iż indykatory efektów są ściśle uzależnione od wskaźników akcji, z których podchodzą $(F * S \# = < I1 * I2)$.

Zakończenie generowania mutex'ów dla wszystkich stanów oraz akcji oznacza ukończenie generowania kolejnej warstwy grafu przez algorytm. Następnie dochodzi do rekurencyjnego wywołania predykatu *graphplan/4* w celu rozszerzenia grafu o kolejną warstwę. Procedura ta jest powtarzana aż do momentu wygenerowania przez predykat *extract_plan/2* odpowiedniego planu.

4.3 Generowanie grafów

Moduł generowania grafów utworzony w języku programowania **python** przy użyciu biblioteki **graphviz**, zgodnie ze swoją nazwą, przeznaczony jest do generowania grafów na podstawie otrzymanych informacji od algorytmu.

UWAGA: Metody wykorzystywane w generowaniu grafów, jak i późniejszych sekcjach tego rozdziału, zostaną zaprezentowane w formie pseudokodów. Dodatkowo, ze względu na prostotę bądź dużą objętość jeśli chodzi o linię kodu, niektóre metody zostały zawarte jedynie w formie opisowej. Kompletne kody źródłowe znajdują się na płycie CD dołączonej do niniejszej pracy w katalogu **sources** (patrz Dodatek A).

4.3.1 Struktura pliku tekstowego

```
1 StartLevel: [pusty(a)/1,na(a,2)/1,pusty(4)/1,pusty(6)/1]
2 A: zostan(pusty(a))
3 Precondition: [pusty(a)]
4 Effects: [pusty(a)]
5 A: idz(a,2,4)
6 Precondition: [pusty(a),pusty(4),na(a,2)]
7 Effects: [na(a,4),pusty(2),~na(a,2),~pusty(4)]
8 A: idz(a,2,6)
9 Precondition: [pusty(a),pusty(6),na(a,2)]
10 Effects: [na(a,6),pusty(2),~na(a,2),~pusty(6)]
11 A: zostan(pusty(a))
12 Precondition: [pusty(a)]
13 Effects: [pusty(a)]
14 A: zostan(na(a,2))
15 Precondition: [na(a,2)]
16 Effects: [na(a,2)]
17 A: zostan(pusty(4))
18 Precondition: [pusty(4)]
19 Effects: [pusty(4)]
20 A: zostan(pusty(6))
21 Precondition: [pusty(6)]
22 Effects: [pusty(6)]
23 Mutex: [pusty(6),na(a,6)]
24 Mutex: [zostan(pusty(6)),idz(a,2,6)]
25 Mutex: [pusty(6),~pusty(6)]
26 Mutex: [zostan(pusty(6)),idz(a,2,6)]
27 Mutex: [pusty(4),na(a,4)]
28 Mutex: [zostan(pusty(4)),idz(a,2,4)]
29 Mutex: [pusty(4),~pusty(4)]
30 Mutex: [zostan(pusty(4)),idz(a,2,4)]
31 Mutex: [na(a,2),pusty(2)]
32 Mutex: [zostan(na(a,2)),idz(a,2,6)]
33 Mutex: [na(a,2),~na(a,2)]
34 Mutex: [zostan(na(a,2)),idz(a,2,6)]
```

Rysunek 4.3: Urywek przykładowego pliku tekstowego wygenerowanego przez algorytm

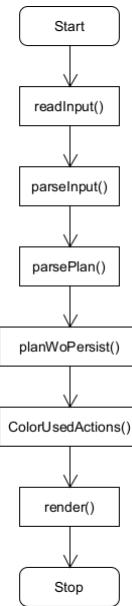
Algorytm na każdym etapie kreowania grafu planującego dokonuje zapisu do pliku tekstowego. Zapisywane są takie informacje jak poziom stanów, akcji czy mutexy. Każda linia pliku rozpoczyna się sygnaturą, po której występuje symbol :, który pełni rolę separatora. Na tej podstawie algorytm dokonuje klasyfikacji informacji występującej po separatorze. Poniżej znajduje się tabela z sygnaturami jak i ich objaśnieniami:

Sygnatura	Objaśnienie
<i>StartLevel</i>	Warunki początkowe przedstawionego świata
<i>A</i>	Obecnie rozpatrywana akcja
<i>Precondition</i>	Warunki zajścia wyżej wymienionej akcji
<i>Effects</i>	Efekty wyżej wymienionej akcji
<i>Mutex</i>	Pary stanów bądź akcji w relacji wykluczającej
<i>ActionLevel</i>	Zbiór wszystkich akcji dla danej warstwy
<i>StateLevel</i>	Zbiór wszystkich stanów dla danej warstwy
<i>ChosenActions</i>	Zbiór akcji wybranych przez planer dla danej warstwy
<i>ChosenStates</i>	Zbiór stanów zdeterminowanych przez wybrane akcje

Tablica 4.1: Tabela sygnatur linii w pliku tekstowym

4.3.2 Graf pełny

Na podstawie otrzymanego pliku tekstowego moduł *parsePlanFULL.py*, w którym zaimplementowana została klasa *Graph*, rozpoczyna generowanie grafu. Pierwszym krokiem jest zainicjalizowanie obiektu klasy *Graph* co jest wykonywane przez warstwę graficzną. Następnie przy pomocy metody *run_all()* dochodzi do uruchomienia procesu generowania grafu. Na powyższym diagramie przepływu należy zauważyć, iż moment oznaczony symbolem *Start* odnosi się do momentu uruchomienia przez warstwę graficzną wyżej wskazanej metody. Po uruchomieniu generatora, moduł wczytuje plik by następnie dokonać jego parsowania.



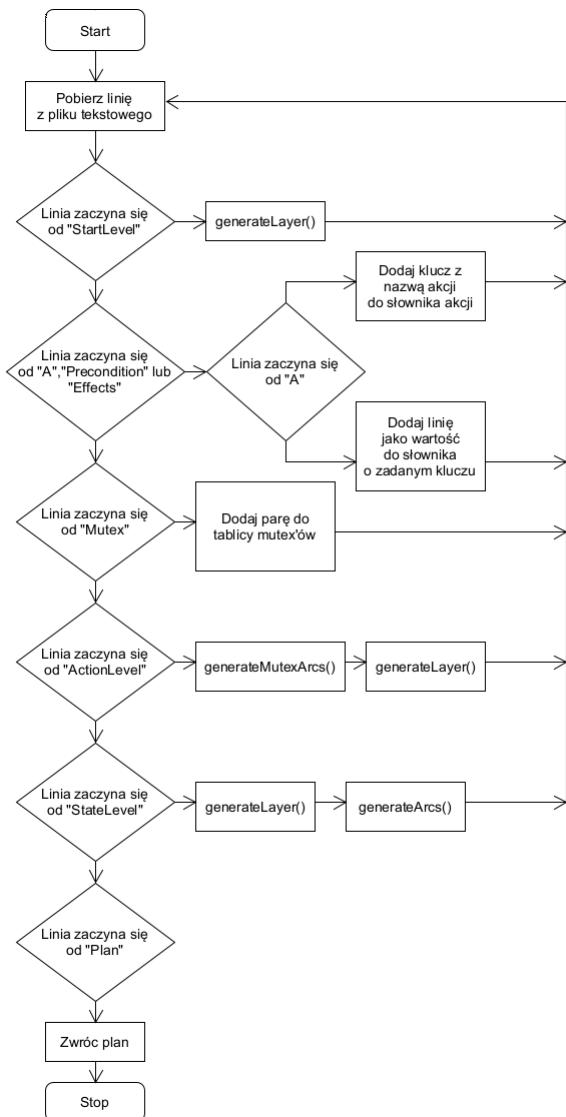
Rysunek 4.4: Diagram przepływu dla generowania pojedynczego grafu pełnego

Funkcja *parse_input()***Pseudokod 4.1:** Schemat działania funkcji *parse_input()*

Input: plik tekstowy *Filename*

```

1 current_level_actions ← Dict{}
2 prev_action ← String()
3 mutex_states ← List[]
4 foreach linia ∈ Filename do
5   name, variables ← linia;
6   variables ← parse(variables);
7   if name = StartLevel then
8     └ generateLayer(variables);
9   if name = A then
10    └ prev_action ← variables;
11    Dodaj prev_action jako klucz do słownika current_level_actions;
12   if name = Precondition then
13    └ Dodaj variables jako warunki zajścia w słowniku current_level_actions dla prev_action;
14   if name = Effects then
15    └ Dodaj variables jako efekty w słowniku current_level_actions dla prev_action;
16   if name = Mutex then
17    └ Dodaj variables do mutex_states;
18   if name = ActionLevel then
19    └ generateMutexArcs(mutex_states);
20    └ mutex_states ← [];
21    └ generateLayer(variables);
22   if name = StateLevel then
23    └ generateArcs(current_level_actions);
24    └ generateLayer(variables);
25    └ current_level_actions ← {};
26   if name = Plan then
27    └ return variables;
  
```



Rysunek 4.5: Szczegółowy diagram przepływu dla parsowania pliku tekstowego

Funkcja w zależności od aktualnej sygnatury zmienia swoje działanie. Należy zauważać, iż wszystkie akcje, a co za tym idzie ich efekty oraz warunki zajścia są zbierane w strukturze określonej mianem **Dict**, czyli tablicy asosjacyjnej (inna nazwa: słownik). Dane w tej strukturze są przechowywane w formie par (*klucz, wartość*). Dokładniej rzecz biorąc, *current_level_action* składa się ze słownika słowników. Kluczem zewnętrznego słownika jest akcja (przykład: *idz(a,1,2)*), następnie kluczem wewnętrznego słownika są dwie wartości: *Precondition* oraz *Effects*, do których w formie wartości dodawane są odpowiednio warunki zajścia jak i efekty zgodnie z nomenklaturą STRIPS. Dodatkowo funkcja składa się z dodatkowej listy o nazwie *mutex_states()*, która zbiera wszystkie relacje wzajemnego wykluczania w danej warstwie grafu.

Funkcja parsująca *parse()* jest *wyrażeniem regularnym*, które konwertuje jeden duży ciąg znaków na listę pomniejszych względem separatora przecinka (textit,). To co należy zauważać, to fakt, iż wyrażenie regularne zostało skonstruowane w taki sposób, aby nie dokonywało podziału linii względem przecinków znajdujących się w nawiasach. Gdyby taka funkcjonalność nie została zaimplementowana, napis *idz(a,2,3)* zostałby podzielony na listę napisów: "[*idz(a,"2","3)*]".

```
variables = re.split(',(?![^()]*\\))',variables)
```

Kod źródłowy 18: Implementacja parsowania pliku tekstowego

Wynik działania ów parsera dla przykładowego napisu "[pusty(3),na(a,3)]" to "[[pusty(3)",na(a,3)]].

Kolejnym krokiem jest uruchomienie funkcji odpowiedzialnych za generowanie grafu w momencie powstawania się odpowiedniej sygnatury. Ów funkcje to odpowiednio: *generateLayer()*, *generateArcs()* oraz *generateMutexArcs()*.

Metoda *generateLayer()* ma za zadanie wygenerowanie poziomu stanów na podstawie otrzymanej listy stanów. Robi to, dodając do istniejącej instancji grafu dostarczonej dzięki bibliotece **graphviz**, kolejnego klastra, który w grafie symbolizowany jest poprzez szarą ramkę.

```
def generateLayer(self,name,variables,g,level,type):
    layer_name = 'cluster_' + name
    with g.subgraph(name=layer_name) as layer:
        layer.attr(color='gray')
        layer.attr(label=name)
        for item in variables:
            layer.node(item+str(level),item, shape=type)
```

Kod źródłowy 19: Implementacja funkcji *generateLayer()*

Każdy z klastrów musi posiadać swoje unikalne imię. Generowane jest ono na takiej samej zasadzie jak nazwy poziomu stanów oraz akcji, poprzez konkatenację słów kluczowych oraz wartości iteratora, który symbolizuje aktualną warstwę grafu planującego.

Metoda *generateArcs()* odpowiada za utworzenie krawędzi między stanami a akcjami. Iterując po zbiorze akcji dla danej warstwy łączy warunki zajścia z akcją oraz akcję z jej efektami. W przypadku, gdy nazwa akcji zawiera w sobie frazę *zostań* generowana jest przerywana linia, symbolizująca akcję podtrzymującą. Krawędzie ciągłe charakteryzuje się tym, iż strzałka składa się z linii ciągłej. Graf planujący jest grafem acyklicznym skierowany, co powoduje, iż każda z krawędzi zakończona jest strzałką- symbolizuje to skierowanąłość, którą można interpretować jako jednostronne połączenie.

Metoda *generateMutexArcs()* odpowiada za utworzenie krawędzi między stanami bądź akcjami, które są ze sobą w relacji wzajemnie wykluczającej. Efektem działania tej funkcji jest niebieska, przerywana linia łącząca w pary odpowiednie obiekty grafu.

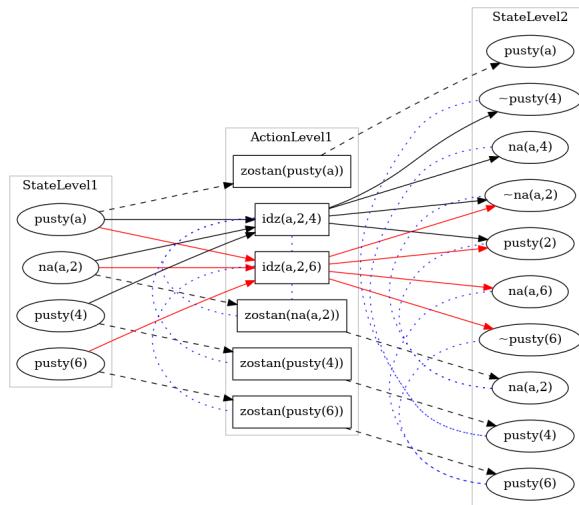
Każdy plik kończy się linią zawierającą plan. Jeśli funkcja *parse_input()* natrafi na rzeczoną linię, parsuje ją i zwraca jej zawartość jako utworzony przez algorytm **plan**.

Oznaczenie planu na grafie

Po zakończeniu działania metody `parse_input()` główna jednostka sterująca przepływem działania generatora grafu przechodzi do funkcji, której zadaniem jest rozbicie planu na warstwy oraz akcje, która w danej warstwie należy wykonać, aby uzyskać wskazany cel. Realizowane jest to przy pomocy funkcji `parsePlan()` oraz wyrażenia regularnego (kod źródłowy 18). Dodatkowo w celu wyeliminowania nieczytelnych dla człowieka informacji z listy akcji eliminowane są akcje podtrzymujące (w odróżnieniu od algorytmu człowiek wie, iż brak któreś z komponentów w akcjach danej warstwy skutkuje brakiem zmiany jego stanu). Wykonywane jest to za pomocą metody `planWoPersist`, która eliminuje wszystkie elementy z planu zawierające frazę `zostań`. Następnie program przechodzi do ostatniego etapu: zaznaczania akcji wchodzących w skład planu poprzez zmianę kolorów na czerwonych wszystkich krawędzi, które z ów akcją są połączone. Implementacja tego mechanizmu została wykonana w ramach funkcji `colorUsedActions()` i w swoim działaniu jest bardzo prostą wyszukuje warunki zajścia, jak i efekty dla wskazanej akcji i zmienia kolory krawędzi, które je łączą. Ważnym jest, iż ze względu na odfiltrowanie akcji podtrzymujących, barwa ich krawędzi nie zostanie zmieniona w wygenerowanym grafie.

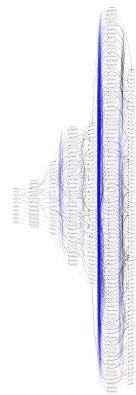
Gdy wszystkie krawędzie zmieniające świat zgodnie z zawartością planu zmienią swój kolor, program wykonuje procedurę `render()`, która generuje graf o zadanych właściwościach. Procedura `render()` pochodzi bezpośrednio z biblioteki `graphviz`.

Efektem działania programu jest następujący graf pełny:



Rysunek 4.6: Przykład grafu pełnego dla prostego przypadku

Jednakże ze względu na liczbę stanów, jakie mogą wchodzić w skład danego poziomu stanów czy akcji oraz na liczbę krawędzi między stanami, graf pełny bardzo szybko staje się nieczytelny dla człowieka, co przedstawia następująca ilustracja:

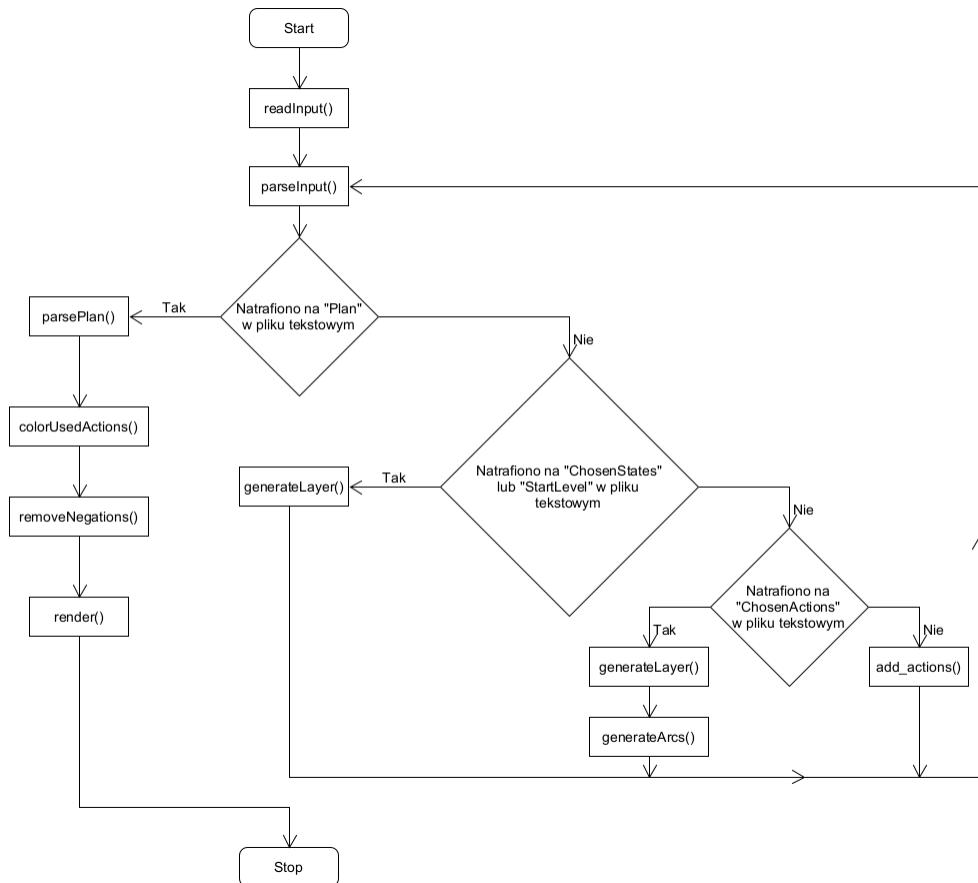


Rysunek 4.7: Przykład grafu pełnego dla złożonego przypadku

Z tego powodu utworzono dodatkową klasę, która generuje podgraf grafu pełnego, który zawiera jedynie istotne z perspektywy człowieka informacje. Taki graf w dalszej części pracy będzie nosił miano *grafu prostego*.

4.3.3 Graf prosty

Graf prosty tworzony jest na podstawie grafu pełnego z wyeliminowaniem pewnych elementów w celu zwiększenia czytelności prezentowanych danych.



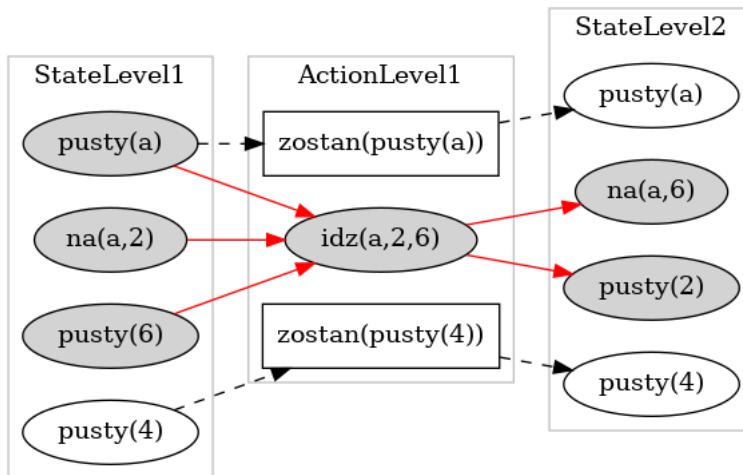
Rysunek 4.8: Diagram przepływu dla generowania pojedynczego grafu prostego

Większość znajdujących się na diagramie funkcji, których implementację można znaleźć w pliku *parseplan-SIMPLIFIED.py* w ramach klasy *SimplifiedGraph*, jest identyczna względem grafu prostego. Główne różnice występują w mechanizmie parsowania planu, ze względu na obiekty, które zostały wyeliminowane na rzecz czytelności grafu. Graf prosty składa się z:

1. Stanu dla każdego obiektu, które są prawdziwe w danej warstwie
2. Akcji dla każdego obiektu w danej warstwie
3. Specjalnego oznaczenia wykorzystywanych akcji aktywnych wraz z oznaczeniem krawędzi, rodem z grafu pełnego

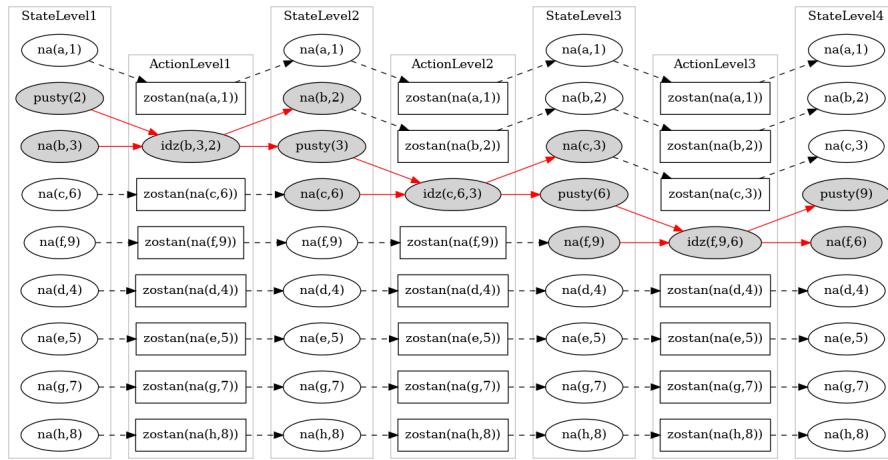
Różnica względem grafu pełnego jest brak generowania mutex'ów, hipotetycznych stanów, w których dany obiekt mógłby się znaleźć oraz efektów usuwających.

Efektem działania programu jest następujący graf:



Rysunek 4.9: Przykład grafu prostego dla prostego przypadku

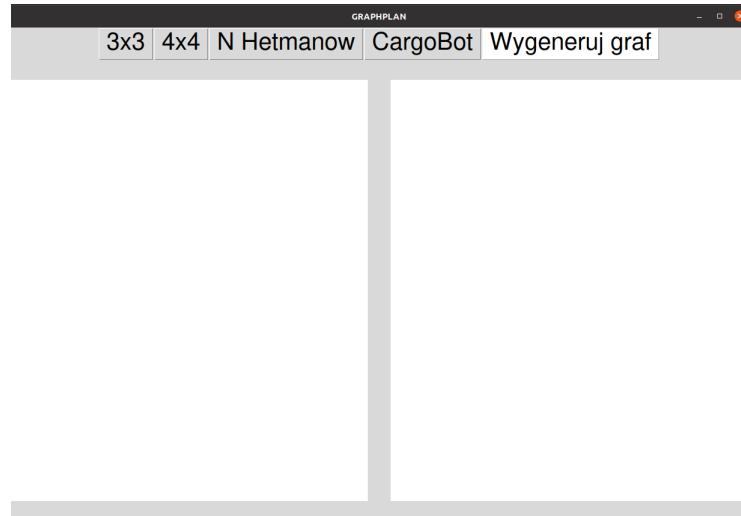
Dla przypadku złożonego czytelność planu wzrosła diametralnie przy drobnej utracie wiedzy o planie. Jednakże, z perspektywy człowieka wykonującego wskazany plan na podstawie grafu, występowanie hipotetycznych stanów jest całkowicie zbędny, istnienie negatywnych efektów funkcji jest dla człowieka naturalnym do wydedukowania, a brak oznaczonych mutex'ów wynika z faktu, iż w skład grafu wchodzą jedynie akcje, które należy wykonać aby otrzymać dany cel. Poniższy przykład przedstawia graf dla złożonego przypadku:



Rysunek 4.10: Przykład grafu pełnego dla złożonego przypadku

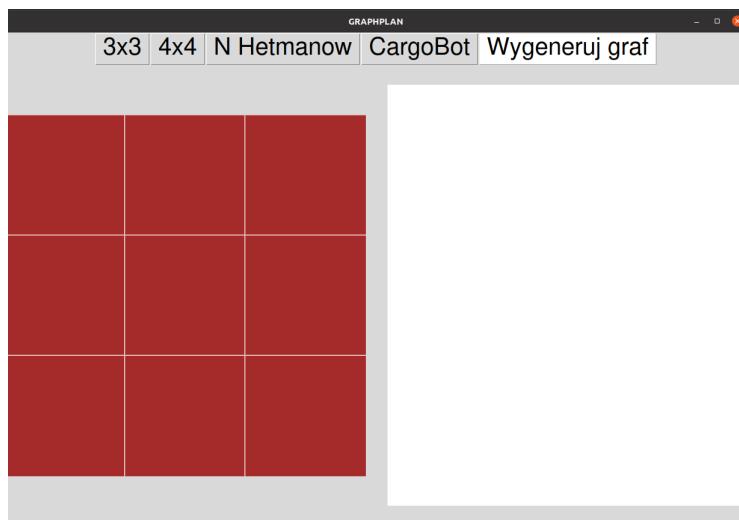
4.4 Interfejs użytkownika

Interfejs użytkownika wykonany w języku **python** stanowi spoiwo, łącząc w sobie wygenerowany plan przez algorytm oraz wygenerowane grafy przez odpowiedni moduł. Przy uruchomieniu programu dochodzi do utworzenia instancji klasy *GUI()*, której zadaniem jest utworzenie okna oraz zapewnienie poniżej opisanych funkcjonalności. Po uruchomieniu pliku *gui.py* (preferowana ścieżka uruchomienia to linia komend, przy użyciu komendy `python3 gui.py`, patrz ??). pojawia się następujące okienko:

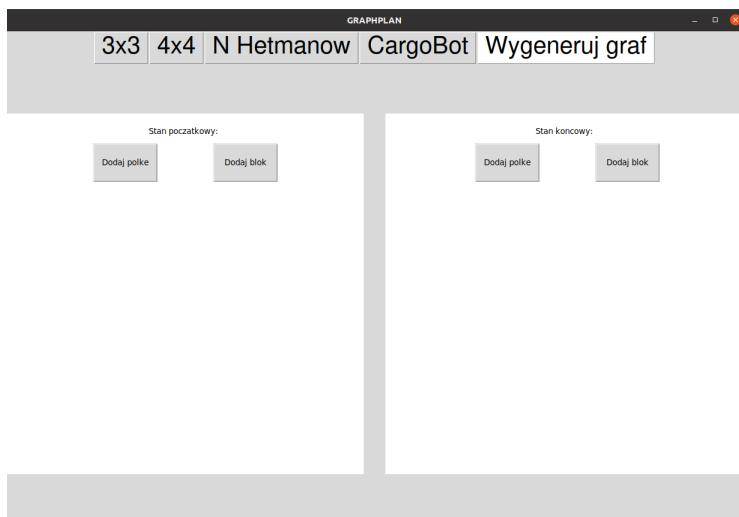


Rysunek 4.11: Okno startowe programu

W górnym panelu znajdują się przyciski, które odpowiadają odpowiednim światom. Sygnatury znajdujące się na przyciskach zostały dokładniej przedstawione w sekcji odpowiedzialnej za testy algorytmu (3x3 i 4x4: 6.1, CargoBOT: 6.2, Osiem Hetmanów: 6.5). Po wybraniu jednej z opcji, użytkownik zostaje przeniesiony w obszar roboczy związany z wybranym otoczeniem.



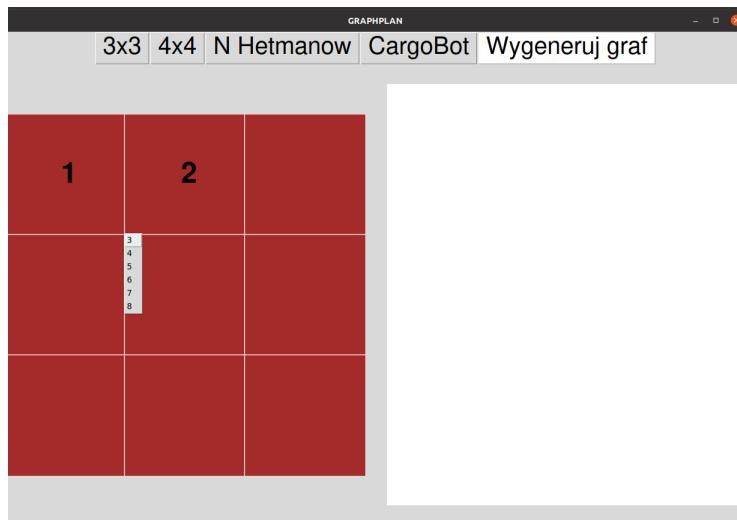
Rysunek 4.12: Widok po wybraniu opcji "3x3"



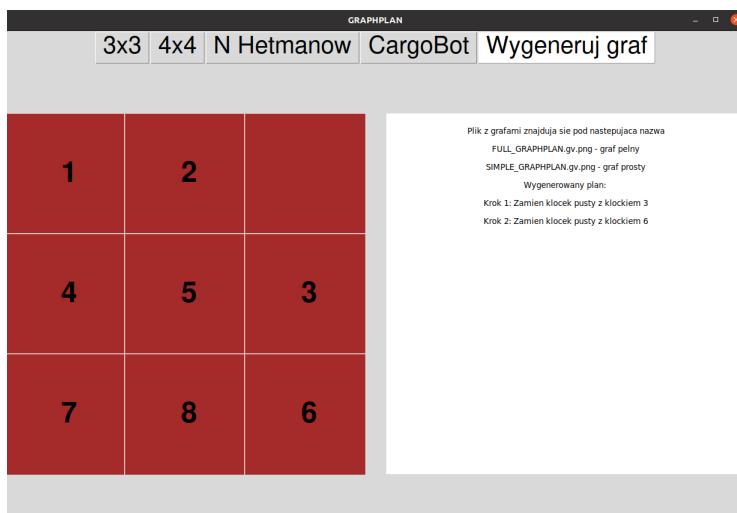
Rysunek 4.13: Widok po wybraniu opcji CargoBOT

Powyższe ilustracje przedstawiają dwa typy światów: jeden, w którym cel jest z góry ustalony, więc użytkownik dysponuje jedynie lewym panelem w celu określenia stanu początkowego, oraz drugi, w którym użytkownik samodzielnie definiuje warunki początkowe jak i wymagany cel.

Po zdefiniowaniu odpowiedniego stanu początkowego, przy pomocy przycisku "Wygeneruj graf" dokonywane jest generowanie grafu wraz z graficznym przedstawieniem planu z podziałem na kroki:



Rysunek 4.14: Proces definiowania świata



Rysunek 4.15: Stan po wygenerowaniu planu

Odpowiednie grafy generowane są w folderze `graphs` o nazwach wskazanych przez okno wynikowe.

4.5 Uruchomienie algorytmu z linii komend

Zalecanym sposobem uruchomienia algorytmu jest wykorzystanie specjalnie przygotowanego programu graficznego, jednakże implementacja zezwala na wykonywanie planów z linii komend. Ów funkcjonalność została wykorzystana między innymi w testach, których opis można znaleźć w ostatnim rozdziale pracy. Aby uruchomić program z linii komend należy wprowadzić następującą komendę `swipl`, która uruchomi środowisko języka programowania PROLOG. Jeśli użytkownik posiada inną implementację języka, zobowiązany jest do samodzielnego zapoznania się ze sposobem uruchomienia interaktywnego środowiska wewnętrz terminala (więcej w sekcji 5.1)

Definicje świata

Przeglądając plik *graphplan.pl* można zauważyc, iż jest on pozbawiony podstawowych definicji wchodzących w skład opisu STRIPS. Wynika to z faktu, iż uruchamiając program przy pomocy graficznego interfejsu, przy wyborze odpowiedniego świata, generowane są wszystkie niezbędne relacje i dołączane do środowiska, w którym znajduje się kod źródłowy algorytmu. Użytkownik samodzielnie musi zadbać o wprowadzenie odpowiednich informacji o świecie. Może to wykonać w następujący sposób:

- Przed wczytaniem zawartości pliku *graphplan.pl* do środowiska interaktywnego może "dokleić" własne linijki kodu zawierające opis świata, w którym chciałby wygenerować odpowiedni plan. Doklejanie należy wykonać po ostatniej komendzie zaczynającej się słowem *dynamic*. Dodatkowo, definiując własne relacje, których nazwy nie zawierają się w sekcji *dynamic*, użytkownik zobowiązany jest do jej uzupełnienia zgodnie z wzorcem, który znajduje się w pliku źródłowym
- Po wczytaniu zawartości pliku *graphplan.pl* użytkownik może manipulować relacjami w świecie poprzez wykorzystywanie predykatów *assert/1* bądź *retract/1*.

Wczytywanie pliku w środowisku interaktywnym SWI-PROLOG można wykonać na dwa sposoby, poprzez zastosowanie nawiasów kwadratowych, wewnątrz których znajduje się nazwa pliku ([*graphplan*]), bądź przy użyciu predykatu *consult/1*. Oba sposoby są sobie równoważne. Opis wczytania pliku zakłada, iż środowisko interaktywne zostało uruchomione w tym samym katalogu, w którym znajduje się kod źródłowy algorytmu.

Uruchomienie algorytmu

Zakładając poprawność definicji świata przez użytkownika należy zaopatrzyć algorytm w dwa istotne aspekty: stan początkowy oraz stan końcowy. Stan początkowy wprowadzany jest przy pomocy predykatu *initial_state/1*, który przyjmuje listę stanów prawdziwych przed rozpoczęciem działań. Następnie należy wywołać procedurę *call_plan/2* według następującego schematu

$$\text{call_plan}(A, \text{Plan}). \quad (4.1)$$

gdzie A oznacza listę stanów docelowych. Przykładowe uruchomienie algorytmu z poziomu linii komend:

```
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- [graphplan].
true.

?- assert(initial_state([na(b,1),na(c,2),na(f,3),na(a,4),na(e,5),pusty(6),na(d,7),
,na(g,8),na(h,9)])).

?- call_plan([na(a,1),na(b,2),na(c,3),na(d,4),na(e,5),na(f,6),na(g,7),na(h,8),pu
sty(9)],Plan).
Plan = [[idz(f, 3, 6)], [idz(c, 2, 3)], [idz(b, 1, 2)], [idz(a, 4, 1)], [idz(d,
7, 4)], [idz(g, 8, 7)], [idz(h, 9, 8)]].

?- ■
```

Rysunek 4.16: Zrzut ekranu prezentujący przykładowe uruchomienie algorytmu z linii komend

W pliku *worlds.pl* znajdują się definicje światów wykorzystywanych w ramach badań. Każdy ze światów odzielony jest komentarzem jednoliniowym wraz z oznaczeniem świata (np. CargoBOT). Kod znajdujący się między sygnaturami komentarza wieloliniowego przedstawia definicję wskazanego świata. Dodatkowo każdy ze światów zawiera kilka przykładów, aby oswoić użytkownika ze specyficznym mechanizmem wprowadzania danych z poziomu linii komend.



Rozdział 5

Instalacja i wdrożenie

UWAGA: Poniższy opis przedstawia sposób instalacji odpowiednich pakietów dla komputerów korzystających z systemu operacyjnego **Linux**, a dokładniej- dystrybucji **Ubuntu**. Użytkownik chcąc zainstalować aplikację wraz z jej komponentami na komputerze z innym systemem operacyjnym zobowiązany jest do samodzielnego zapoznania się ze wszystkimi komendami bądź mechanizami umożliwiającymi instalację wskazanych pakietów.

5.1 Instalacja pakietu SWI-Prolog

Korzystając z dystrybucji Linuxa o nazie Ubuntu, wystarczającą czynnością do poprawnej instalacji pakietu SWI-Prolog jest uruchomienie następującej komendy z poziomu linii komend wraz z prawami administratora

```
sudo apt install swi-prolog-core
```

Kod źródłowy 20: Instalacja pakietu SWI-PROLOG z poziomu linii komend

W momencie, w którym komputer zakończy pobieranie oraz instalację pakietu wprowadzenie komendy *swipl* powinno spodoować uruchomienie interaktywnego interpretera języka PROLOG. Jeśli powyższa czynność zakończyła się sukcesem, komputer jest gotowy do uruchomienia kodu źródłowego algorytmu oraz rozpoczęcia pracy nad kreowaniem odpowiednich planów

5.2 Instalacja języka Python

?? Większość dystrybucji Linuxa posiada wbudowany w sobie język programowania python. Z reguły można to zweryfikować poprzez wpisanie komendy

```
python3 --version
```

Kod źródłowy 21: Komenda sprawdzająca wersję zainstalowanego języka python

Należy zauważać, iż wszystkie komponenty zostały napisane dla wersji języka python 3.x. Użytkownik korzystając ze starszych wersji może spotkać się z anomaliemi negatywnie wpływającymi na funkcjonowanie aplikacji, dlatego zaleca się korzystanie ze wskazanej powyżej wersji. Do poprawnego uruchomienia aplikacji wymagane są następujące biblioteki

- Tkinter



- graphviz
- PIL
- pyswip

Poniżej znajdują się odpowiednie komendy, który użycie z poziomu linii komend zagwarantuje poprawne uruchomienie aplikacji:

```
sudo apt install python3-tk
pip3 install graphviz
pip3 install Pillow
pip3 install pyswip
```

Kod źródłowy 22: Instalacja odpowiednich bibliotek dla języka python

Przed uruchomieniem powyższych komend użytkownik winien posiadać zainstalowany pakiet *pip*. Jeśli pobieranie wskazanych bibliotek zakończy się błędem, należy uprzednio wykonać następującą komendę: `textttsudo apt install python3-pip`. Całą aplikację również uruchamia się z poziomu linii komend. Po pobraniu odpowiednich plików, w folderze `sources` znajduje się plik `gui.py`, który zawiera kod rozruchowy interfejsu użytkownika. Będąc we wskazany katalogu należy uruchomić terminal i wprowadzić następującą komendę:

```
python3 gui.py
```

Kod źródłowy 23: Uruchomienie interfejsu użytkownika

Jeśli operacja zakończy się sukcesem, użytkownik powinien ujrzeć okienko identyczne do tego, które zostało przedstawione w ramach sekcji 4.4

5.3 Dokumentacja

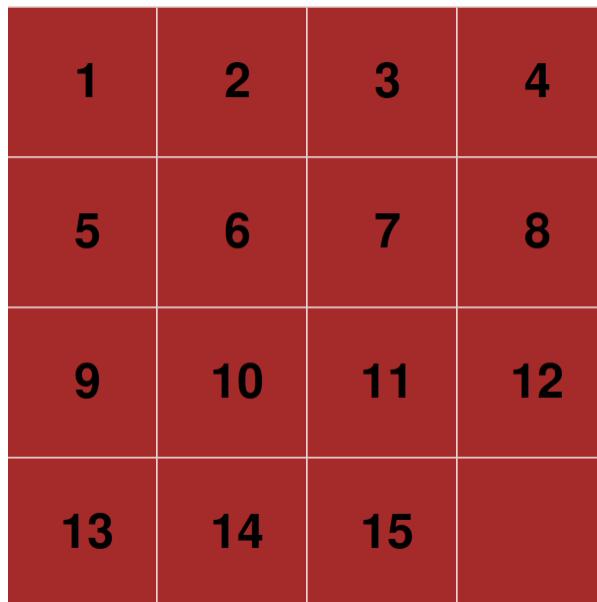
Rozdział 6

Testy

6.1 15

6.1.1 Wprowadzenie

Piętnastka (fr. taquin), znana również w Polsce o nazwie *przesuwanka*, w zapisie często podawana przy pomocy liczbowego odpowiednika (15) jest grą w formie specyficznej układanki, której powstanie datuje się na koniec XIX wieku. Składa się ona z 15 klocków oraz ramki, pierwotnie drewnianej. Ramka zaprojektowana została specjalnie z myślą o pozostawieniu jednego wolnego miejsca, aby móc w łatwy sposób przesuwać klocki sąsiadujące z miejscem pustym. Celem gry jest ułożenie klocków w określony sposób, najczęściej w porządku rosnącym czytając od lewej do prawej rzędami, z określonego stanu początkowego. Częstym zabiegiem stosowanym przez twórców układanki jest konwersja liczb na części obrazka, aby zachęcić do gry młodszych odbiorców.



Rysunek 6.1: Wygląd układanki piętnastki wygenerowany przy pomocy zaimplementowanej w ramach pracy warstwy graficznej

14	3	4	2
1	10	5	13
11	6	9	7
8		15	12

Rysunek 6.2: Losowa rozwiązywalna permutacja układanki

Rysunek 6.3: Piętnastka w formie obrazkowej. Źródło: <http://mypuzzlecollection.blogspot.com/2012/08/mc-escher-birds-fish-and-turtles.html>

6.1.2 Teoria

W 1878 roku amerykański wynalazca gier i zagadek (między innymi zagadek szachowych) **Samuel Loyd** ze względu na swój fach nie przeszedł obojętnie koło piętnastki proponując ułożenie układu rosnącego z pozycji, która od początkowej różniła się pozycjami jedynie dwóch klocków numerowanych odpowiednio 14 i 15 [2]. Problem ów stał się na tyle popularny, iż została wyznaczona nagroda 1000 dolarów dla osoby, której udałoby się znaleźć prawidłowe rozwiązanie przygotowanego przez Pana Samuela problemu.

Niemogliność ułożenia problemu przez bardzo długi czas doprowadziła do pierwszych poważniejszych rozwiazań nad z pozoru trywialną łamigłówką. Efektem prac matematyków było parę zaskakujących wniosków, które ostatecznie doprowadziły do udowodnienia, iż wyżej przedstawiona łamigłówka jest nierozwiązywalna.

Lemat 6.1 *Nie wszystkie ustawienia początkowe piętnastki są możliwe do rozwiązania. [5]*

Wynika to z faktu, iż dla układanki o parzystych rozmiarach (w tym przypadku układanka jest rozmiarów 4x4) rozwiązywalne są jedynie ułożenia o parzystej liczbie inwersji. Zagadka Pana Lyod'a jest ustawieniem nieparzystym jeśli chodzi o inwersje. Prowadzi to do następującego wniosku:

Wniosek 6.1 *Istnieje $\frac{16!}{2} = 10461394944000$ rozwiązywalnych ustawień.*

Dodatkową ciekawostką istotną z perspektywy wykonywanych testów jest maksymalna liczba posunięć, którą należy wykonać, aby z rozwiązywalnego stanu osiągnąć wcześniej wyznaczony cel. Mianem **boskiej liczby** w odniesieniu do przesuwanki określą się największą liczbę posunięć, którą trzeba osiągnąć, aby rozwiązać najtrudniejsze ułożeniem początkowe. Przy pomocy matematyki naukowcy odnaleźli najtrudniejsze ustawienia oraz obliczyli ów liczbę, co zaprezentowano w następującym lemacie:

Lemat 6.2 Boska liczba dla 15-elementowej przesuwanki wynosi 80. [1]

Oznacza to, iż maksymalna liczba kroków algorytmu w żadnym wypadku nie powinna przekroczyć liczby 80.

W trakcie poniżej opisanego testu sprawdzono plany przesuwania odpowiednich klocków, aby w jak najmniejowej możliwej liczbie ruchów otrzymać odpowiedni stan końcowy, przy okazji sprawdzono osiągnięcia czasowe jak i porównano otrzymane wyniki z popularnymi herustykami spersonalizowanymi pod rozwiązywanie ów układanki.

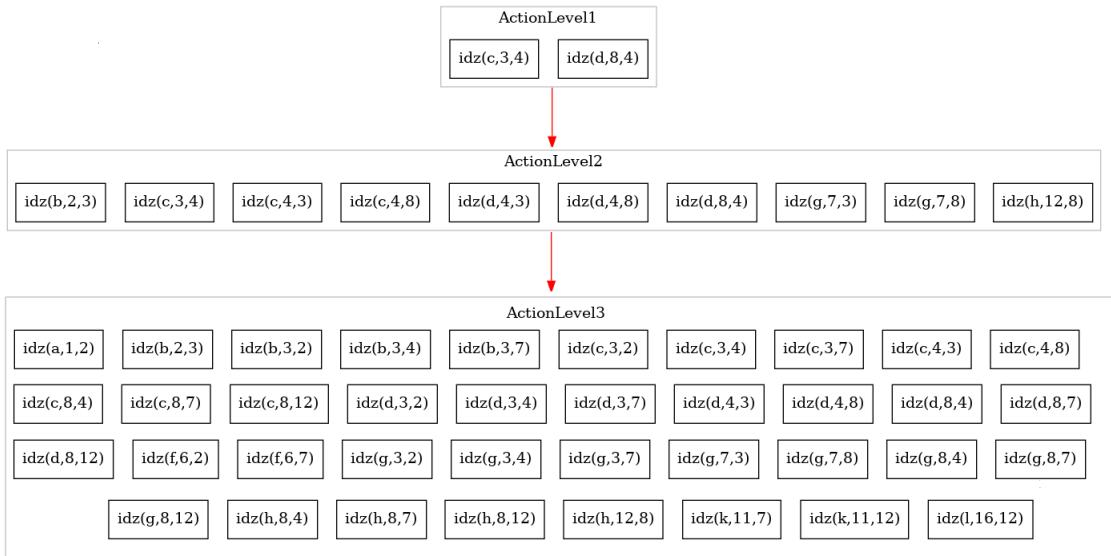
6.1.3 Przykład

Na podstawie następującego przykładu zostanie przedstawiony schemat rozwiązywania układanki przez algorytm:

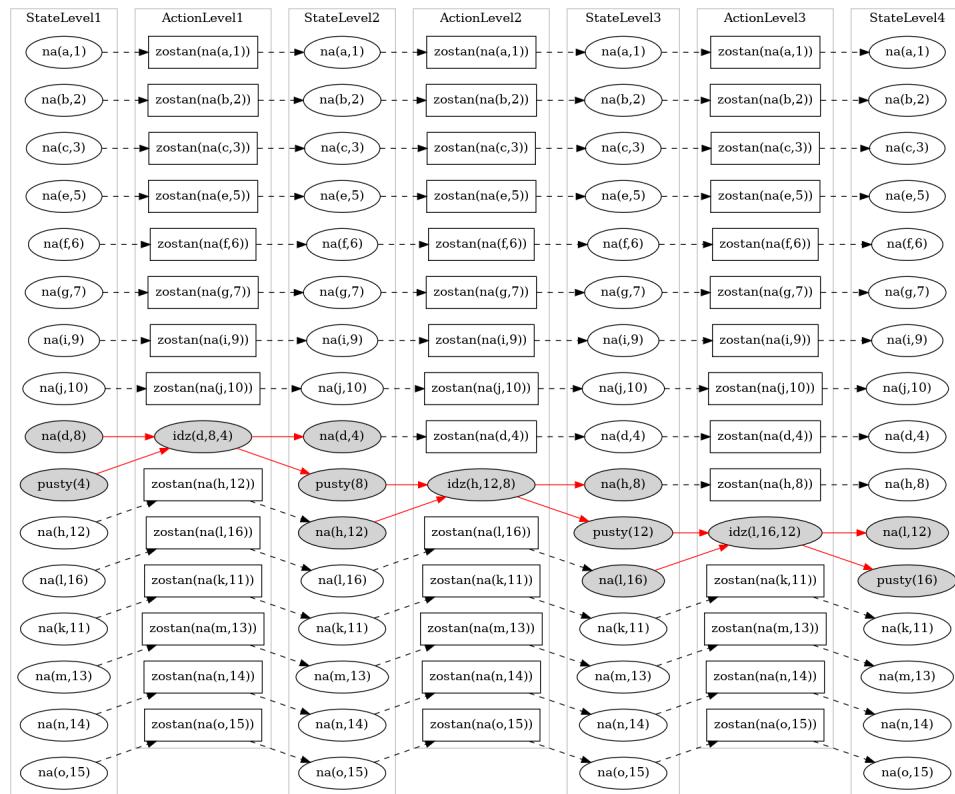
1	2	3	
5	6	7	4
9	10	11	8
13	14	15	12

Rysunek 6.4: Przykładowe startowe ułożenie przesuwanki

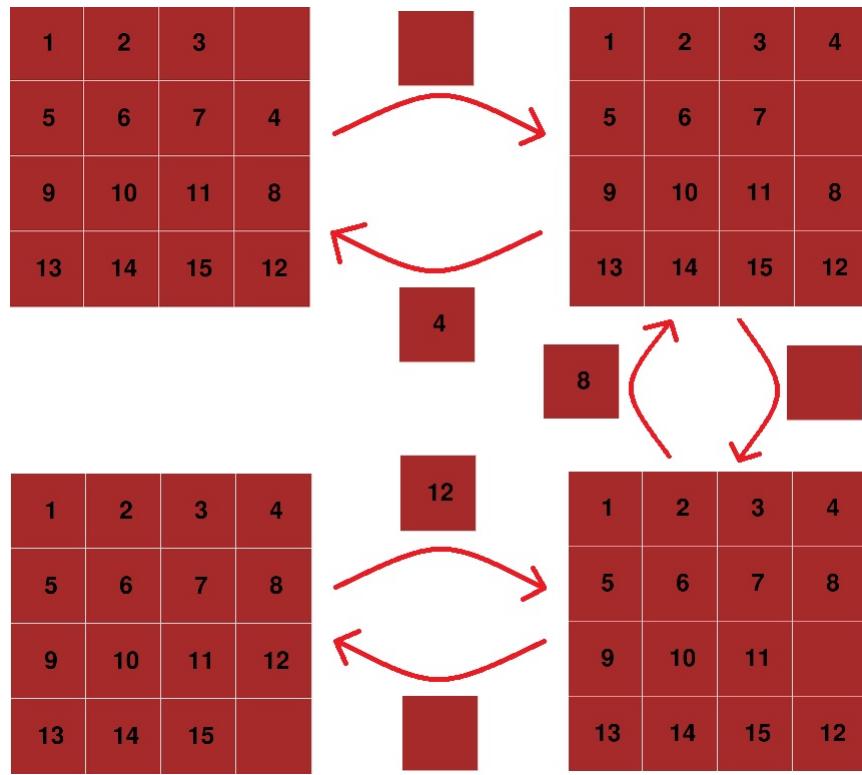
Algorytm buduje graf na podstawie jedynej zdefiniowanej w problemie akcji: przesuwania klocka. Robi to aż do momentu, gdy kafelki nie będą ułożone w wcześniej zdefiniowanej kolejności. Dla zdefiniowanego powyżej przykładu w pierwszym kroku algorytm ma jedynie dwie możliwości akcji aktywnych: zamiana klocka pustego z klockiem o numerze 3 lub klockiem o numerze 4. Na tej podstawie generuje kolejny poziom stanów. Dla załączonego przykładu optymalnym rozwiązaniem jest odpowiednio zamiana pustego kafelka z kafelkami: 4,8,12, co zostało poprawnie wyznaczone przez GRAPHPLAN. Poniżej przedstawiono zbiory akcji przeanalizowane przez algorytm w danym kroku jak i uproszczony graf planujący.



Rysunek 6.5: Akcje rozpatrywane przez algorytm w danym kroku



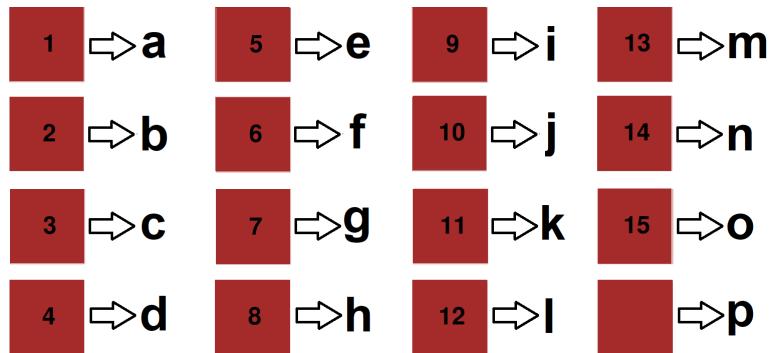
Rysunek 6.6: Uproszczony graf planujący wygenerowany przez algorytm GRAPHPLAN przedstawiający stan każdego kafelka w danej warstwie. Węzły wypełnione kolorem szarym obrazują stany, które są warunkami zajścia jak i efektami wykonywanej w danej warstwie akcji



Rysunek 6.7: Obrazowe rozwiązywanie na podstawie wygenerowanego grafu

6.1.4 Szczegóły implementacyjne

Ważnym jest, aby przedstawić omawiany świat zgodnie z wytycznymi ustalonymi przez język **STRIPS**. Z tego powodu należy dokładnie określić każdą istniejącą relację, które zostaną użyte do definiowania stanów, akcji, warunków początkowych oraz celów. Pierwszą określona relacją będzie dwuargumentowa relacja *na*, której opis został już zawarty w sekcji 2.3. W ramach, relacja *na(A,B)* informuje algorytm o tym, iż klocek *A* znajduje się na pozycji *B*. Następnie należy zdefiniować każdy z klocków. Wykorzystano do tego jednoargumentowy predykat o nazwie *robot(R)*. Ze względu na to, iż stałe w prologu oznaczane są przy pomocy małych liter każdy z klocków zawiera swoje odzworowanie w literach. Poniższa ilustracja przedstawia proces mapowania klocka na literę:



Rysunek 6.8: Przyporządkowywanie klockom odpowiednich liter

W tym momencie należy wprowadzić rozróżnienie między klockiem a pozycją. Pozycje na planszy są stałe, numerowane rzędami od lewej do prawej, natomiast klocki mogą dynamicznie zmieniać swoją pozycję.

Oznacza to tyle, iż numer klocka nie jest jednoznaczny z numerem pozycji. Zrozumienie tego aspektu jest kluczowe, gdyż konwersja liczb na litery może prowadzić do zmieszania. Dodatkowo należy zdefiniować relację $pusty(A)$, która informuje o tym, iż dana pozycja jest pusta, lub formalniej, jest okupowana przez klocek o pustej sygnaturze.

Jedyną akcją aktywną zdefiniowaną w ramach przesuwanki jest akcja $idz(R,A,B)$, która przesuwa klocek R z pozycji A na pozycję B . Zgodnie z wprowadzeniem literowych oznaczeń klocków, akcję $idz(a,4,8)$ należy rozumieć jako przesunięcie klocka z numerem 1 z pozycji 4 na pozycję 8 (czyli pionowo o jedną pozycję w dół w ostatniej kolumnie). Dla każdej akcji należy zdefiniować jej warunki zajścia jak i efekty. W przypadku przesuwanki dokonano tego przy pomocy predykatów $preconditions/2$ oraz $effects/2$. Poniżej znajdują się ich implementacje w języku programowania PROLOG:

```
%preconditions(+Action,-States)
preconditions(zostan(P),[P]).  
  
preconditions(idz(R,A,B), [na(R,A), pusty(B)]) :-  
    robot(R),  
    adjacent(A,B).
```

Kod źródłowy 24: Implementacja predykatu $preconditions/2$ dla przesuwanki

```
%effects(+Action,-States)
effects(zostan(P),[P]).  
  
effects(idz(R,A,B), [na(R,B),pusty(A),~na(R,A),~pusty(B)]).
```

Kod źródłowy 25: Implementacja predykatu $effects/2$ dla przesuwanki

Należy zwrócić uwagę na fakt, iż każdy z predykatów występuje w dwóch wersjach, jedna odpowiada za akcję podtrzymującą, w skład której zawsze wchodzi jeden warunek zajścia oraz jeden efekt, oraz drugą, która odpowiada akcji aktywnej. W przypadku predykatu $preconditions/2$ wykorzystaną relację $adjacent(A,B)$ która zachodzi w sytuacji, gdy pozycja A sąsiaduje z pozycją B . Definicja sąsiedztwa jest identyczna względem definicji wprowadzonej w 2.4.

```
%eadjacent(+From, +To)
adjacent(A,B) :-  
    n(A,B)  
;  
    n(B,A).
```

Kod źródłowy 26: Implementacja predykatu $adjacent/2$

Gdzie relacja $n(A,B)$ (n od angielskiego słowa neighbor, oznaczającego sąsiada) przedstawia relację sąsiedztwa pozycji A z pozycją B . Poniżej przedstawiono rozkład relacji sąsiedztwa dla każdej z pozycji wchodzącej w skład przesuwanki.



```
n(1,2). n(1,5).
n(2,1). n(2,6). n(2,3).
n(3,2). n(3,7). n(3,4).
n(4,3). n(4,8).
n(5,1). n(5,6). n(5,9).
n(6,5). n(6,2). n(6,7). n(6,10).
n(7,6). n(7,3). n(7,8). n(7,11).
n(8,7). n(8,4). n(8,12).
n(9,5). n(9,10). n(9,13).
n(10,9). n(10,6). n(10,11). n(10,14).
n(11,10). n(11,7). n(11,12). n(11,15).
n(12,11). n(12,8). n(12,16).
n(13,9). n(13,14).
n(14,13). n(14,10). n(14,15).
n(15,14). n(15,11). n(15,16).
n(16,15). n(16,12).
```

Kod źródłowy 27: Modelowanie relacji sąsiedztwa

Ostatnim aspektem, bez którego wykonywanie działań w opisywanym świecie jest wprowadzenie stanów niespójnych, takich jak znajdowanie się jednego bloczka na dwóch różnych pozycjach, bądź okupowanie jednej pozycji przez dwa różne klocki. Zostały to wykonane przy pomocy predykatu *inconsistant/2*

```
inconsistent(+State1,+State2)
inconsistent(G, ~G).
inconsistent(~G,G).
inconsistent(na(R,C1),na(R,C2)) :-
    C1 \== C2.

inconsistent(na(_,C),pusty(C)).
inconsistent(pusty(C),na(_,C)).
inconsistent(na(R1,C),na(R2,C)) :-
    R1 \== R2.
```

Kod źródłowy 28: Implementacja predykatu inconsistent/2

Po zdefiniowaniu powyższych predykatów, algorytm jest gotowy do generowania odpowiednich planów mających za zadanie ułożenie układanki z dowolnej rozwiązywalnej pozycji początkowej.

6.1.5 Wyniki

UWAGA: Testy czasowe zaprezentowane w poniższych tabelach tyczą się osiągów samego algorytmu. Oznacza to, iż na czas wykonywania prób wyłączone zostały wszystkie poboczne funkcjonalności takie jak generowanie grafu, czy prezentowanie rozwiązania w formie graficznej. Wykonanie poniższych badań w aplikacji może skutkować innymi wynikami, zwykle dłuższymi. Ponadto testy czasowe obarczone są nieprawidłowościami związanymi z działaniem innych, równoległych procesów w trakcie uruchamiania programu. Należy mieć to na uwadze przy potencjalnej próbie odtwarzania badań.

W kodach źródłowych (patrz Dodatek A) w katalogu *sources* znajduje się plik o nazwie *15_tests.txt*, w którym przedstawione są dokładne stany początkowe, dla których algorytm został uruchomiony, całkowity plan wygenerowany przez algorytm oraz liczbę wnioskowań jak i wykorzystany czas dla każdego z przypadków. Ze względów objętościowych, w niniejszej pracy zostaną zamieszczone jedyne najważniejsze informacje.

Przesunięcia	Test1	Test2	Test3	Test4	Test5
1	0.009	0.009	—	—	—
2		0.026	—	—	—
3		0.126	—	—	—
4		0.519	—	—	—
5		1.214	—	—	—
6		1.847	—	—	—
7		5.539	—	—	—
8		9.082	—	—	—
9		10.964	—	—	—
10		30.142	—	—	—

Tablica 6.1: Przykładowe czasy dla wybranych przykładów

Przesunięcia	Test1	Test2	Test3	Test4	Test5
1	0.009	0.009	—	—	—
2		0.026	—	—	—
3		0.126	—	—	—
4		0.519	—	—	—
5		1.214	—	—	—
6		1.847	—	—	—
7		5.539	—	—	—
8		9.082	—	—	—
9		10.964	—	—	—
10		30.142	—	—	—

Tablica 6.2: Przykładowa liczba wnioskowań dla wybranych przykładów

Przesunięcia	Wnioskowania	Czas
1		0.009
2		0.026
3		0.126
4		0.519
5		1.214
6		1.847
7		5.539
8		9.082
9		10.964
10		30.142

Tablica 6.3: Średnia liczba wnioskowania oraz czasów wykonania

Przez *przesunięcia* należy rozumieć liczbę przesunięć, którą należy wykonać, aby wrócić do początkowego stanu. Dla początkowych wartości wskazanych w tabelach zostało to wyznaczone poprzez ręczne przedstawienie kafelków o daną liczbę przesunięć oraz zapisanie otrzymanego stanu przy użyciu nomenklatury opisanej w rozdziale definiującym świat. Zgodnie z powyższym, analizując przykład ?? łatwo zauważyc, iż wymaganą liczbą przesunięć, potrzebną do powrotu do wyjściowego stanu przesuwanki jest liczba **3** (co w dalszej części udało się udowodnić konstruując odpowiedni plan).

6.1.6 Młodsza siostra- ósemka

Ósemka jest inną formą przesuwanki- zamiast tradycyjnego ułożenia 15 klocków w kształcie kwadratu 4 na 4 z jednym wolnym miejscem, w ósemka składa się z osmiu klocków, które ułożone zostały w kształcie kwadratu 3 na 3. Powoduje to zmniejszenie dziedziny, a co za tym idzie- łatwiejsze i szybsze układanie. Dla ósemki wszystkie definicje i własności są identyczne jak dla piętnastki, z drobnym wyjątkiem- dla układanki o osmiu klockach i jednym pustym polu **boska liczba** wynosi nie 80 a 31[7], oraz liczba możliwych ułożen wynosi

$$\frac{8!}{2} = 181440 \quad (6.1)$$

6.1.7 Wyniki dla 8

Wersja ze zmniejszoną liczbą klocków została poddana identycznym testom jak piętnastka. Poniżej przedstawione zostały rezultaty przeprowadzonych badań:

Przesunięcia	Test1	Test2	Test3	Test4	Test5
1	0.009	0.009	—	—	—
2		0.026	—	—	—
3		0.126	—	—	—
4		0.519	—	—	—
5		1.214	—	—	—
6		1.847	—	—	—
7		5.539	—	—	—
8		9.082	—	—	—
9		10.964	—	—	—
10		30.142	—	—	—

Tablica 6.4: Przykładowe czasy dla wybranych przykładów

Przesunięcia	Test1	Test2	Test3	Test4	Test5
1	0.009	0.009	—	—	—
2		0.026	—	—	—
3		0.126	—	—	—
4		0.519	—	—	—
5		1.214	—	—	—
6		1.847	—	—	—
7		5.539	—	—	—
8		9.082	—	—	—
9		10.964	—	—	—
10		30.142	—	—	—

Tablica 6.5: Przykładowa liczba wnioskowań dla wybranych przykładów

Przesunięcia	Wnioskowania	Czas
1		0.009
2		0.026
3		0.126
4		0.519
5		1.214
6		1.847
7		5.539
8		9.082
9		10.964
10		30.142

Tablica 6.6: Średnia liczba wnioskowania oraz czasów wykonania

6.1.8 Wnioski

Dla zdefiniowanego świata GRAPHPLAN poprawnie generuje plany. Empiryczne sprawdzenie wykazało, iż faktycznie są to plany optymalne. Ponadto dla mniejszych przykładów generowanie planów trwa stosunkowo krótko, jednakże w sytuacji, gdy dojdzie do spłaszczenia grafu planującego, co dla przesuwanki z 15 klockami dzieje się w okolicach 10 iteracji, wydajność algorytmu znacznie spada. Podobne efekty zostały zaobserwowane dla układanki z 8 klockami. Porównanie z specjalnie utworzonymi na potrzeby układanki heurystykami pokazały, iż są one znaczco szybsze niż algorytm planujący, jednakże duża zaleta GRAPHPLAN'u jest uniwersalność, gdyż można go stosować do innych problemów, heurystyki natomiast rozwiązuje problem jedynie dla danej układanki.

6.2 CargoBot

6.2.1 Wprowadzenie

CargoBOT (Twórcy: **Two Lives Left**) jest grą dostępną w platformie dystrybucji cyfrowej o nazwie **App Store**, która znajduje się na urządzeniach wyprodukowanych przez firmę **Apple**. Głównym celem gry jest przestawienie skrzynek z pozycji początkowej na przedstawioną pozycję końcową przy użyciu ramienia, które może poruszać się w prawo, w lewo oraz opuszczać na dół. Podczas opuszczania, jeśli w ramieniu znajduje się skrzynka to zostanie opuszczona, natomiast jeśli ramię nie trzyma żadnej skrzynki podniesie pierwszą na jaką natrafi. Przypatrując się bliżej powyższemu opisowi, nie sposób nie zauważyc połączenia przedstawionego świata z językiem opisowym STRIPS. Z tego powodu kolejnym testem możliwości GRAPHPLAN'u będzie rozwiązywanie zagadek przedstawionych przez aplikację **CargoBOT**. **UWAGA:** W grze CargoBOT celem użytkownika jest zaprojektowanie algorytmu przy pomocy specjalnego panelu do rozwiązywania łamigłówki. Ze względu na to, iż celem testu jest przedstawienie optymalnego planu otrzymującego stan końcowy ze stanu początkowego, wynik algorytmu nie będzie bezpośrednio rozwiązaniem korzystającym z nomenklatury aplikacji.

6.2.2 Szczegóły implementacyjne

Przed rozpoczęciem generowania planu, podobnie jak w przypadku poprzednich testów, należy rozpocząć modelowanie świata zgodnie z wytycznymi języka opisu STRIPS. Pierwsze zaimplementowane predykaty to *block/1* oraz *place/1*. Fakt *block(a)* oznacza skrzynkę z etykietą a, natomiast *place(1)* oznacza miejsce z etykietą 1. Ze względu na ograniczenia językowe, każdy z bloków oraz każde z miejsc musi mieć unikatową etykietę. Spowoduje to, iż nie wszystkie łamigłówki z aplikacji będą możliwe do reprodukcji ze względu na to, iż paczki w grze rozróżnialne są od siebie jedynie ze względu na *kolor*. Dodatkowo przez pojęcie miejsca rozumie się przestrzeń, na której może leżeć bloczek. Miejsca można wyobrażać sobie jakie półki, bądź platformy, które są od siebie odzielone pewną przeszkołą bądź wolną przestrzenią. Przykład ?? przedstawia reprezentację miejsc w formie czerwonych platform.

To co odróżnia opis świata dla aplikacji CargoBOT od przesuwanki to fakt, iż liczba skrzynek oraz platform może zmieniać się dynamicznie w zależności od świata. Z tego powodu za każdym razem należy upewnić się, iż wszystkie obiekty zostały opisane w języku algorytmu przy pomocy wyżej przedstawionych predykatów.

Jedyną akcją rozpatrywaną jest przesuwanie przy użyciu ramienia skrzynki z jednej platformy na drugą. Akcja ta będzie nazwana identycznie jak w przypadku przesuwanki przy pomocy czasownika *idz*. Trójargumentowa relacja *idz(S,A,B)* oznacza przesunięcie skrzynki *S* z platformy *A* na platformę *B*. Poniżej przedstawiono implementację warunków zajścia jak i efektów przedstawionej akcji:

```
%preconditions(+Action,-States)
preconditions(zostan(P),[P]).  
  
preconditions(idz(Block,From,To), [pusty(Block),pusty(To),na(Block,From)]) :-  
    block(Block),  
    object(To),  
    To \== Block,  
    object(From),  
    From \== To,  
    Block \== From.
```

Kod źródłowy 29: Implementacja predykatu preconditions/2 dla CargoBOT'a

```
%effects(+Action,-States)
effects(zostan(P),[P]).  
  
effects(idz(X,From,To),[na(X,To),pusty(From),^na(X,From),^pusty(To)]).
```

Kod źródłowy 30: Implementacja predykatu effects/2 CargoBOT'a

W przypadku akcji przesunięcia warto zaznaczyć, iż może zaistnieć sytuacja, w której najlepszą drogą uzyskania stanu końcowego jest umieszczenie jednej paczki na drugiej. Stąd znane już czytelnikowi predykaty *na/2* oraz *pusty/1* w tym przypadku będą się odnosić do skrzynek, ale również do platform. Przyglądając się ciele predykatu *preconditions/2* należy zauważać, iż dochodzi do sprawdzenia, czy to co znajduje się w zmiennej *Block* jest na pewno typu *block*, oraz czy zmienne *From* oraz *To* są obiektami świata. Predykat *object/1* realizowany jest w następujący sposób:

```
%object(+Variable)
object(X) :-  
    place(X)  
    ;  
    block(X).
```

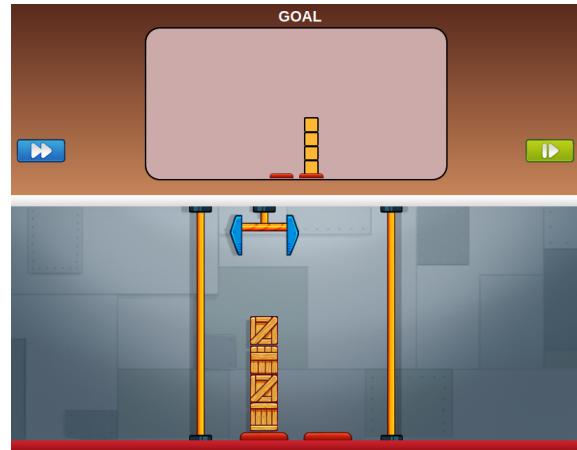
Kod źródłowy 31: Implementacja predykatu object/1

Zgodnie z powyższą definicją platformy również są obiektami, dlatego należy dokonać dodatkowego sprawdzenia, aby uniemożliwić powstanie takich anomalii jak umieszczanie całej platformy na jednej ze skrzynek.

Dodatkowe sprawdzanie wykonywane w ramach warunków zajścia mają za zadanie uniknięcie sytuacji, w której planer chciałby przenieść skrzynkę na samą siebie. Implementacja relacji wzajemnie wykluczającej odbyła się w sposób identyczny jak dla 15. Użyty został predykat *inconsistent/2*. W powyższy sposób ukończono implementację świata z gry CargoBOT.

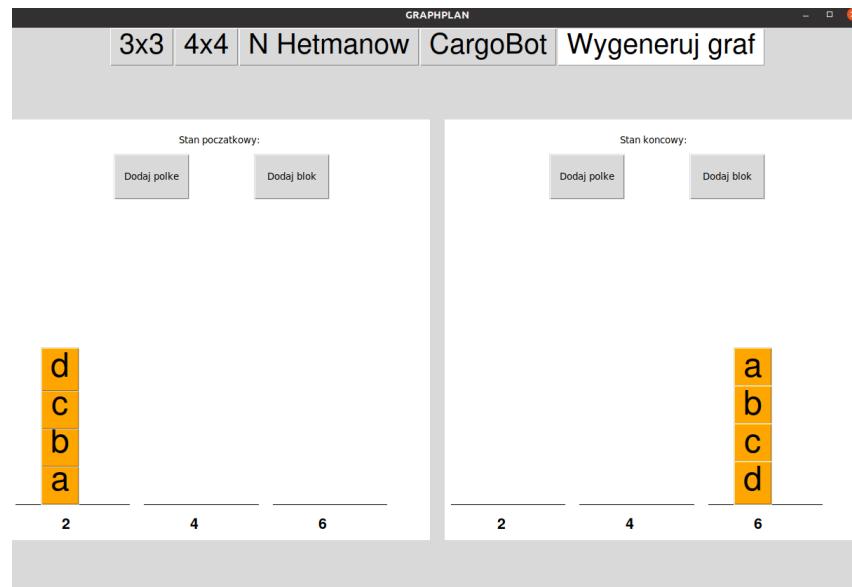
6.2.3 Przykład

Celem algorytmu będzie rozwiązanie następującego problemu:



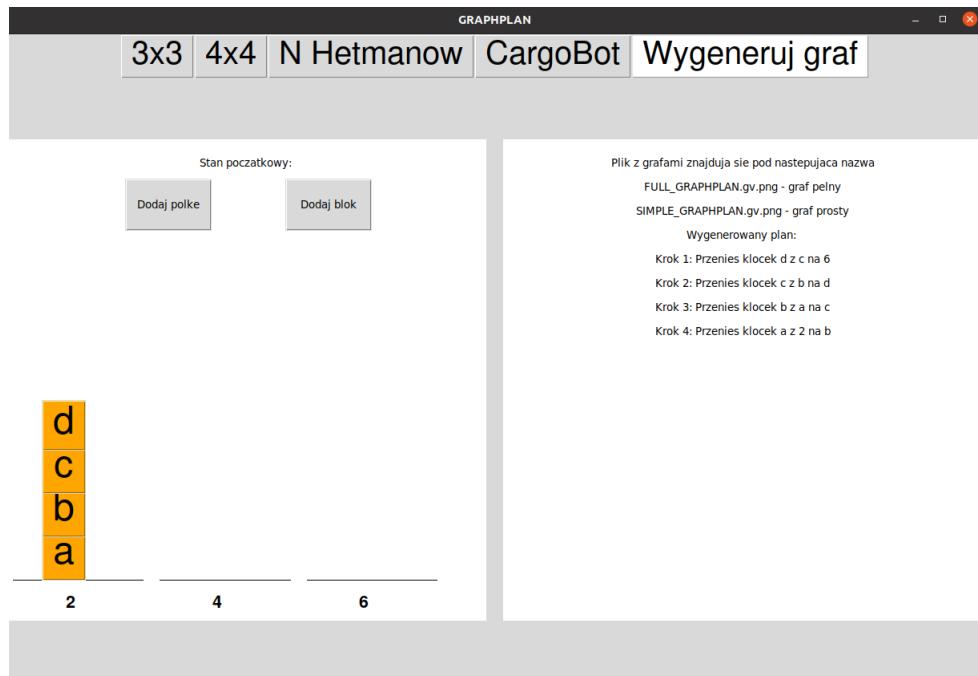
Rysunek 6.9: Przykładowa łamigłówka, źródło zdjęcia: <https://i4ds.github.io/CargoBot/?state=3>[ostatni dostęp:06.12.2022]

Sytuacja ta została odzworowana w aplikacji graficznej w następujący sposób:



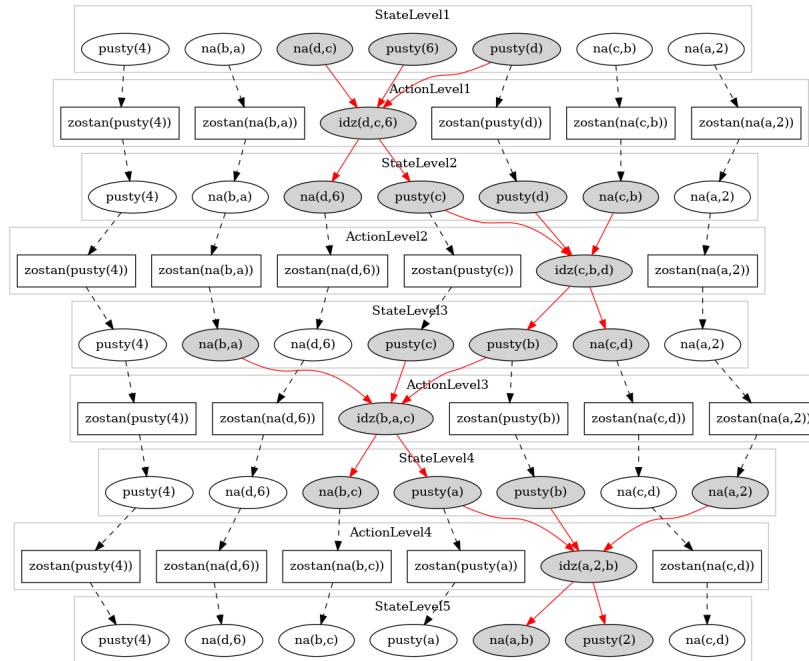
Rysunek 6.10: Reprezentacja łamigłówki w aplikacji

Po naciśnięciu przycisku *Wygeneruj graf* widok zmienia się w następujący sposób:



Rysunek 6.11: Rozwiążanie w formie listy kroków zaprezentowane w aplikacji

Poniżej zamieszczono wygenerowany graf prosty dla rozpatrywanego przypadku:



Rysunek 6.12: Reprezentacja łamigłówki w aplikacji



6.2.4 Wyniki

6.2.5 Wnioski

6.3 Przemieszczanie w przestrzeni

6.3.1 Wprowadzenie

6.3.2 Przykład

6.3.3 Szczegóły implementacyjne

6.3.4 Wyniki

6.3.5 Wnioski

6.4 Wieża Hanoi

6.4.1 Wprowadzenie

6.4.2 Przykład

6.4.3 Szczegóły implementacyjne

6.4.4 Wyniki

6.4.5 Wnioski

6.5 Osiem Hetmanów

6.5.1 Wprowadzenie

6.5.2 Przykład

6.5.3 Szczegóły implementacyjne

6.5.4 Wyniki

6.5.5 Wnioski

Podsumowanie

Główny cel pracy, którym była implementacja algorytmu GRAPHPLAN wraz z programowaniem ograniczeń, został osiągnięty. Przy pomocy nowego podejścia do programowania udało się ułatwić proces implementacji jak i przyśpieszyć czas tworzenia planu przez algorytm. Użytkownik może skorzystać z wbudowanych światów w ramach interfejsu graficznego, bądź uruchomić program z poziomu linii komend przy okazji definiując własne środowisko pracy. Na żądanie użytkownika generowane są odpowiednie grafy reprezentujące plan, jak i schemat transformacji świata z wyszczególnieniem sytuacji, w jakiej znajduje się każdy stan początkowy na dowolnym etapie realizacji planu. W ramach sprawdzenia funkcjonalności wykonano testy na prostych, lecz obrazowych przykładach. Na ich podstawie można wywnioskować, iż GRAPHPLAN, zgodnie ze swoim założeniem generuje plany, które są optymalne, czyli składają się jak najmniejszej liczby kroków. Wykonując testy na popularnej *przesuwance* można było zauważyc, iż rozszerzanie grafu planującego znacznie wpływa na długość wykonywania algorytmu. Wprowadzenie świata z mniejszą liczbą stanów, w postaci ósemki, dokonało lekkiej poprawy sytuacji.

Algorytm posiada szereg możliwości pozwalających na rozszerzenie jego funkcjonalności oraz zwiększenie efektywności generowania planów. W trakcie implementacji można zastosować mechanizm *podwójnego szukania*, który miałby za zadanie układać plan symultanicznie korzystając z mechanizmu cofania oraz, na wzór ludzki, dynamicznie zmieniając świat próbując w ten sposób uzyskać wyznaczony cel. Dodatkowo warty uwagi jest fakt, iż w niektórych sytuacjach można zrezygnować z gwarancji najkrótszego możliwego rozwiązania na rzecz szybszego generowania algorytmu. Ponadto algorytm w swojej istocie bazuje głównie na relacji wzajemnego wykluczania. Istnieje możliwość, iż w zdefiniowanych światach istnieją inne informacje, które prowadziły do szybszego wyszukiwania odpowiedniego planu.

Dalsze możliwe kierunki rozwoju oprogramowania to między innymi generowanie większej liczby grafów o różnych własnościach, które w dokładniejszy sposób prezentowałyby użytkownika esencję algorytmu, rozbudowa warty graficznej algorytmu pod kątem estetycznym jak i pod kątem liczby światów w jakich użytkownik może generować plany.



Bibliografia

- [1] K. F. Adrian Brünggera, Ambros Marzettab, J. Nievergelt. The parallel search bench zram and its applications. <http://www.iro.umontreal.ca/~gendron/Pisa/References/BB/Brungger99.pdf> [ostatni dostęp: 06.12.2022].
- [2] A. F. Archer. A modern treatment of the 15 puzzle. <https://www.cs.cmu.edu/afs/cs/academic/class/15859-f01/www/notes/15-puzzle.pdf> [ostatni dostęp: 06.12.2022], 1999.
- [3] M. L. F. Avrim L. Blum. Fast planning through planning graph analysis, 1997.
- [4] I. Bratko. *Prolog Programming for Artificial Intelligence*. Pearson, 2012.
- [5] C. Calabro. Solving the 15-puzzle. <https://cseweb.ucsd.edu/~ccalabro/essays/15-puzzle.pdf> [ostatni dostęp: 06.12.2022], 2005.
- [6] T. W. Francesca Rossi, Peter van Beek. *Handbook of Constraint Programming*. 2006.
- [7] A. Reinefeld. Complete solution of the eight-puzzle and the benefit of node ordering in ida*. <https://www.ijcai.org/Proceedings/93-1/Papers/035.pdf> [ostatni dostęp: 06.12.2022].
- [8] N. J. N. Richard E. Fikes. Strips: A new approach to the application of theorem proving to problem solving, 1971.
- [9] N. P. Russell Stuart. *Artificial Intelligence A Modern Approach (4th Edition)*. Pearson, 2020.



Załącznik A

Zawartość płyty CD

W tym rozdziale należy krótko omówić zawartość dołączonej płyty CD.

